N° d'enregistrement au C.N.R.S.
A.O. 9.243

THESE

présentée à

L'UNIVERSITE PARIS VI

pour l'obtention du

DOCTORAT D'ETAT ES SCIENCES

Spécialité: INFORMATIQUE

par M.onsieur Gérard GUIHO

Sujet de la thèse: Organisation de Mémoires, Influence d'une structure et étude d'optimisation

Soutenue le 3 Décembre 73 devant la Commission d'examen

composée de: Président J.C. SIMON

Examinateurs L. NOLIN
J.C. BOUSSARD

Invités : G. SALTON
G. BOULAYE

N° d'enregistrement
au C.N.R.S.

A.O. 9.243

THESE

présentée à

L'UNIVERSITE PARIS VI

pour l'obtention du

DOCTORAT D'ETAT ES SCIENCES

Spécialité: INFORMATIQUE

par M.onsieur Gérard GUIHO

Sujet de la thèse: Organisation de Mémoires, Influence d'une structure et étude d'optimisation

Soutenue le 3 Décembre 73 devant la Commission d'examen

composée de: Président J.C. SIMON

Examinateurs L. NOLIN

J.C. BOUSSARD

Invités : G. SALTON

G. BOULAYE

UNIV.
LEIDEN,
BSL.

Je remercie particulièrement le Professeur J.C. SIMON pour toute l'aide et les conseils qu'il n'a cessé de me prodiguer depuis le début de ce travail. La plupart des idées de cette thèse sont venues après de nombreuses et fructueuses discussions, je le remercie vivement.

Je remercie Messieurs les professeurs L. NOLIN et G. SALTON qui ont bien voulu s'intéresser à mon travail ainsi que Messieurs les Professeurs J.C. BOUSSARD et G. BOULAYE qui, avec eux, ont accepté de former le jury.

Je remercie aussi Melle LOYO et Messieurs ATTIAS, BACCHUS, CASSO, GOVAERT, PARIS, ROTH, Etudiants de 3ème Cycle ou Elèves de l'Ecole Polytechnique pour l'aide qu'il m'ont apporté.

Enfin je remercie sincèrement Madame B. RIVIERE pour le très bon travail réalisé dans la frappe de cette thèse ainsi que l'équipe de Madame HANNEQUIN qui en a réalisé la publication.

I. INTRODUCTION

A. Caractéristiques des ordinateurs 7

1. Utilisation
2. Temps
3. Mémoire

B. Contraintes 8

C. Méthodes

D. Influence de la structure 9

II. GENERALITES

A. Algorithme ou mémoire ? 13

B. Coefficients importants 18

1. Coefficient α

a. α_a
b. α_b
c. α_c

2. Coefficient β
3. Coefficient γ
4. Coefficient γ'
5. Coefficient de création
6. Coefficient de retrait δ

C. Contraintes d'utilisation 27

1. Méthodes
2. Utilisation

III. METHODES D'ORGANISATION

A. Prétraitement 36

1. Mesure du gain de mémoire 36
2. Propriétés du gain 40
3. Relation avec les grammaires 45
4. Relation avec l'information mutuelle 48
5. Construction d'un prétraitement 50

- définition des opérateurs
- composition d'opérateurs
- gain résultant
- algorithme de construction

B. Algorithmes de recherches 61

1. Méthode séquentielle 61
2. Méthode dichotomique 62
3. Méthode des listes 65
4. Méthode directe 72
5. Hash Coding 75

a. Présentation, généralités 75

b. Etude générale 80

- choix de h_0
- résolution des collisions
- facteur de groupage
- mémoire de débordement

c. Mémoires successives 106

1. Problème pratique
2. Problème théorique
3. Résolution approchée

d. Influence de l'adresse 115

1. Débordement-chaînages
2. Mémoires successives

6. Méthode approximative 122

a. Obtention de $f_n(x)$
b. Méthode de rangement
c. Méthode de recherche
d. Résultats
e. Lissage de $f_n(x)$
f. Conclusion

ANNEXE A Méthode approximative appliquée aux Tris 133

ANNEXE B Courbes ; résultats concernant la méthode approximative 153

ANNEXE C Fonctions d'approximation 161

BIBLIOGRAPHIE 167

INTRODUCTION

INTRODUCTION

INTRODUCTION

A. Une des caractéristiques essentielles des ordinateurs est la mémoire, la possibilité de conserver "quelque chose" pour le retrouver plus tard. De ce fait, la plupart des branches de l'informatique se trouvent constamment confrontées avec le problème de ranger un grand nombre de valeurs (on dit aussi variables, informations, clefs, ...) dans une mémoire en vue de les utiliser par la suite.

Il existe trois contraintes très importantes et inévitables à ce problème.

1. L'utilisation qu'on désire faire de la mémoire

Ce point n'est pas à négliger. En effet, rien ne sert de créer une méthode de gestion très fine d'une mémoire utilisant un grand nombre de caractéristiques connues du fichier pour une utilisation banale et faible. C'est souvent un facteur qui est couramment oublié. Ceci est dû sans doute au fait que ce facteur est difficilement quantifiable.

2. Le temps

Les ordinateurs fonctionnent très vite et même de plus en plus vite mais ces temps ne seront jamais assez brefs (la vitesse est limitée par celle de la lumière) pour un grand nombre de problèmes, qu'ils soient scientifiques ou de gestion. La tendance actuelle étant au télétraitement en temps réel, véritable service de l'ordinateur à la mesure de l'homme, on essaye d'effectuer le plus grand nombre d'actions possibles dans un temps de réponse acceptable et le facteur temps ne peut être que renforcé.

3. *La mémoire*

Les mémoires aussi sont de plus en plus grandes mais toujours insuffisantes. L'importance de ce facteur est mis en évidence par les dernières améliorations apportées à la technologie des ordinateurs :

Antémémoires qui sont destinées à diminuer le temps d'accès par une utilisation statistique d'une partie plus rapide et doublée de la mémoire.

Mémoires virtuelles constituées d'une mémoire rapide et de mémoires plus lentes avec une gestion continuelle d'échanges entre les deux types de mémoires. Un système soit cablé, soit programmé ou mixte permet d'amener en mémoire rapide des parties de "mémoire lente" suivant les besoins de l'utilisateur, cet utilisateur n'ayant pas à s'occuper en principe de ces échanges.

Bien entendu, ce système conduit à un temps d'accès mémoire moyen plus élevé et n'exclut nullement une bonne gestion (au moins au niveau du système).

B. Les contraintes de temps et de mémoire sont souvent liées à des contraintes de prix et sont difficilement modifiables. La contrainte d'utilisation est conditionnée, elle, par le but désiré ce qui fait qu'il n'existe pas de méthode universelle et optimale de recherche en mémoire. Il est possible de comparer différentes méthodes mais difficile de proclamer que l'une est meilleure que telle autre.

En fait, chaque utilisation désirée d'une mémoire définit un ordre sur les méthodes qui lui est propre. On peut ensuite comparer les méthodes connues suivant cet ordre et choisir la meilleure.

Ces différents ordonnancements de méthodes utilisent néanmois un certain nombre de critères connus où au moins il existe un certain ensemble de critères (temps d'accès, nombre moyen d'accès, coefficient d'occupation... qui contient les critères propres à chaque méthode suffisants pour exprimer la relation d'ordre. Nous avons en particulier dans cette thèse dégagé et quantifié cet ensemble de critères.

C. Un grand nombre de méthodes d'organisation de mémoires ont été créées en vue de différentes utilisations, depuis les méthodes séquentielles jusqu'aux techniques de "Hash Coding". Très peu de comparaisons réelles ont été effectuées et très souvent, lorsque des comparaisons ont eu lieu, elles ont été faites avec des critères non normalisés entre eux. Nous avons aussi dans cette thèse étudié, dans leurs **principes**, la plupart des méthodes existantes, nous en avons développé certaines et créé d'autres en étudiant, à chaque fois leur influence sur l'ensemble de critères connus. En particulier nous avons du redéfinir la notion de "mémoire pleine" en montrant que la méthode séquentielle n'est pas optimale du point de vue occupation mémoire. Pour cela nous avons défini une quantité minimale de mémoire a priori et en améliorant certaines méthodes montré qu'il était possible de se rapprocher de ce minimum. L'idée importante est que l'adresse ou le numéro d'ordre d'une valeur peut contenir une information importante sur cette valeur.

Une des conséquences importantes de ceci est que le coefficient d'occupation mémoire (Load Factor α) défini dans la littérature n'est qu'un coefficient de comparaison d'occupation mémoire entre méthodes.

D. Intuitivement, il semble que dans tout problème d'organisation de mémoire, la connaissance d'une structure sur l'ensemble de valeurs est importante. La tentation est souvent très grande d'utiliser au maximum cette structure *indépendamment de ce qu'on désire faire de la mémoire*. Nous avons essayé de quantifier l'influence de cette structure comme un gain effectif de mémoire. On s'aperçoit en fait que la connaissance d'une structure ne provoque qu'une économie de mémoire mais n'améliore que très rarement les coefficients de recherche. La structure est surtout importante dans le cas où on désire retrouver une information qu'on ne connaît que partiellement mais il s'agit alors d'un autre problème.

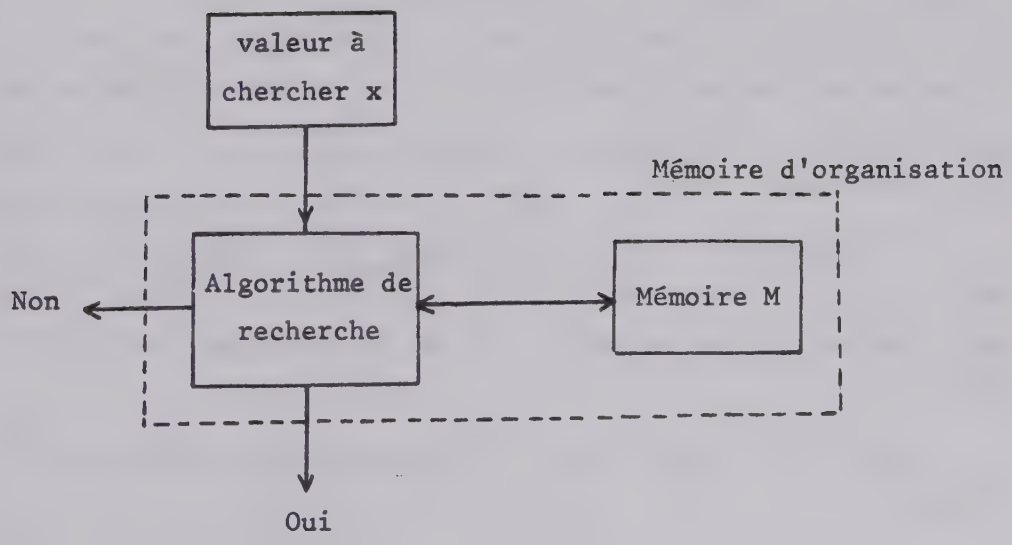
GENERALITES

GENERALITES

A. *ALGORITHME OU MEMOIRE ?*

Une méthode d'organisation de recherche en mémoire peut être considérée comme un algorithme permettant de décider si une valeur x prise parmi un ensemble de N valeurs possibles, est ou n'est pas, dans un sous ensemble X de N.

Cet algorithme peut utiliser une mémoire M contenant une "certaine information" sur l'ensemble X.



La sortie OUI est souvent accompagnée d'un indicatif qui sert de repère de la valeur, soit dans la mémoire, soit dans une autre mémoire de référence (par exemple les enregistrements d'un fichier sont stockés sur disque et les clefs des enregistrements ainsi que les adresses correspondantes constituent l'ensemble X des valeurs à rechercher).

On verra plus tard que dès qu'un système d'organisation commence à être assez élaboré, la mémoire M *n'est pas une représentation directe binaire de l'ensemble X*. L'algorithme de recherche contient lui-même une certaine information sur cet ensemble.

Ce partage entre algorithme de recherche et mémoire n'est d'ailleurs pas très net, l'algorithme étant représenté par un programme qui lui-même est contenu dans une mémoire. On ne peut concevoir une mémoire sans algorithme permettant d'y accéder mais il est moins évident qu'on puisse remplacer toute mémoire (et algorithme de recherche) par un algorithme seul.

Exemple :

Supposons un ensemble N de valeurs représentables chacune sur n bits (c'est déjà une information sur les valeurs).

Supposons que nous disposons d'une machine comportant n registres et permettant une modification d'adresse des instructions de saut absolu.

Soit SAUT ad, i

L'instruction de saut à l'adresse "ad" modifiée par le contenu du registre i. Soit OUI et NON les deux sorties de l'algorithme de recherche auxquelles on accède par les instructions SAUT OUI ou SAUT NON.

Soit x_i le ième bit de la représentation binaire de la valeur x.

Le programme de recherche peut commencer de la façon suivante :

$$\forall i,\ x_i \rightarrow \text{registre } i$$

puis

```
SAUT    ad1, 1
```

Il existe alors deux adresses de destinations possibles suivant la valeur du premier bit de x. Aux adresses ad1 et ad1 + 1 on trouve

```
SAUT    ad11, 2
SAUT    ad12, 2
```

et ainsi de suite, les parties adresses des instructions pouvant éventuellement être remplacées par OUI ou NON.

Si nous supposons l'ensemble X suivant

$$\{X\} = \left\{ \begin{matrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{matrix} \right\}$$

L'algorithme de recherche peut être le suivant après chargement de registres.

```
          SAUT    B1, 1
B1        SAUT    BO2, 2
          SAUT    B12, 2
BO2       SAUT    NON
          SAUT    BO13, 3
BO13      SAUT    NON
          SAUT    BO11, 4
BO114     SAUT    OUI
          SAUT    NON
B12       SAUT    B103, 3
          SAUT    NON
B103      SAUT    B1004, 4
          SAUT    B1014, 4
B1004     SAUT    NON
          SAUT    OUI
B1014     SAUT    OUI
          SAUT    OUI
```

Ceci peut sembler lourd car l'ensemble X est très petit mais peut être intéressant pour un gros fichier.

En fait, cette solution n'est certainement pas la meilleure du point de vue occupation mémoire, on pourrait l'améliorer en remplaçant l'instruction SAUT B1014, 4 par SAUT OUI et en supprimant les deux dernières instructions. On peut aussi réarranger la suite d'instructions mais on pourrait surtout concevoir un interpréteur qui n'aurait besoin que des parties adresses des instructions (puisqu'il s'agit toujours de la même

instruction) et même puisqu'on connaît à tout moment le bit qu'on vient de tester (le numéro du registre qu'on a utilisé) on peut ne conserver que :

	B1
B1	BO2
	B12
BO2	NON
	BO13
BO13	NON
	BO114
BO114	OUI
	NON
B12	B103
	NON
B103	B1004
	OUI
B1004	NON
	OUI

On a ainsi constitué une liste de mémoire à deux pointeurs dont la valeur des cellules est constituée par le choix entre les pointeurs.

Cet exemple simple est important car il montre que puisqu'on peut remplacer une recherche en mémoire par un algorithme seul, *il ne suffit pas de comparer des mémoires pour comparer des méthodes d'organisation* du point de vue de l'occupation de la mémoire.

Ce défaut est courant. Très souvent la place occupée par l'algorithme est négligée et même souvent la place occupée par les pointeurs ou les zones de débordement.

La taille de l'algorithme (la quantité de mémoire nécessaire pour sa représentation) est souvent négligeable devant la taille du fichier. Il semble raisonnable de ne pas tenir compte de cette place occupée mais ce ne peut pas être une règle absolue. Certaines organisations permettent de retrouver une valeur en un ou deux accès au maximum à des mémoires externes telles des disques. Si l'algorithme de recherche est trop important, il peut nécessiter, de par sa taille, d'être segmenté en "overlays" et il perd alors toute efficacité puisque l'appel d'un overlay peut être du même ordre de grandeur (en temps), qu'un accès à la mémoire. Et ceci est indépendant des qualités réelles de l'algorithme, même s'il permet ensuite de retrouver les valeurs en seul accès. (Nous avons supposé que l'algorithme et le fichier étaient rangés sur des mémoires de même type).

B. *COEFFICIENTS DE METHODES IMPORTANTS*

1. *Coefficient d'occupation de la mémoire : α*

Nous définirons ce coefficient comme étant le rapport entre une quantité qui caractérise l'ensemble X des valeurs à ranger et une quantité qui caractérise la mémoire utilisée.

$$\alpha = \frac{\varphi(X)}{\psi(M, M_p)}$$

X ensemble des valeurs à conserver

M mémoire utilisée

M_p mémoire utilisée pour représenter l'algorithme

Il serait logique de prendre pour $\varphi(X)$ un nombre qui soit un minimum compte tenu du fichier, tel que ψ étant mesuré avec une unité identique, le coefficient α soit toujours compris entre 0 et 1. L'usage courant est différent.

a. Les machines ayant une structure de mots, on a commencé, en supposant que la représentation d'une valeur x tient sur un mot, par représenter α, comme étant le rapport entre le nombre d'éléments de l'ensemble X et le nombre de mots de la mémoire.

$\varphi(X)$ = cardinal (X) que nous noterons désormais [X]

$\psi(M, M_p)$ = nombre de mots de M

Les unités sont différentes, il n'y a qu'une relation de compatibilité entre elles mais l'usage d'un tel coefficient α est très répandu. Si on dispose d'un fichier de 15 000 valeurs que l'on range dans une mémoire de 20 000 mots on dira que nous avons une mémoire remplie à 75 % indépendant de la nature des valeurs et du nombre de bits par mots.

b. Aucune comparaison réelle n'étant possible avec ce dernier coefficient, on a essayé de tout ramener à une unité commune très utilisée en informatique : le bit.

$$\varphi(X) = [X] \quad \mathrm{Log}_2 [N]$$

On suppose que chaque valeur x faisant partie de l'ensemble N de $[N]$ éléments, il suffit de $\text{Log}\,[N]$ bits pour la représenter et donc de $[X]\ \text{Log}\,[N]$ bits pour représenter toutes les valeurs de l'ensemble X

$$\psi(M, M_p) = \text{nombre de bits de } M$$

$$\alpha_b = \frac{[X]\ \text{Log}_2\,[N]}{\text{nombre de bits de } M}$$

Ceci aboutit à un coefficient α qui peut être parfois supérieur à 1 (ce n'est toutefois pas très courant). Cette habitude est due à l'historique des méthodes d'organisation de mémoire. La toute première, la plus simple consiste à introduire les valeurs l'une après l'autre dans la mémoire (méthode séquentielle) on ne s'occupe pas du coefficient d'occupation puisqu'on considère (à tort !) que la mémoire est pleine ($\alpha = 1$). Plus tard, d'autres méthodes d'organisation sont apparues et au détriment de cette quantité de mémoire que l'on utilise dans la méthode séquentielle. Il est clair qu'une méthode plus élaborée, améliore les performances mais demande plus de mémoire qu'auparavant. Le coefficient d'occupation en mémoire a une certaine importance ; elle est évaluée en conservant pour référence la quantité de mémoire utilisée dans la méthode séquentielle.

Il est cependant facile de trouver des exemples montrant que ce coefficient, défini de cette façon, peut être plus grand que 1.

Supposons X constitué de $[X]$ éléments pris parmi N.
La quantité $\varphi(X)$ est donc évaluée à $[X]\ \text{Log}_2\,[N]$.

Comme organisation, prenons la méthode suivante :

Considérons un tableau TAB de [N] mots de 1 bit. Si l'élément x appartient à X nous avons TAB(x) = 1 sinon TAB(x) = 0.

La quantité de mémoire utilisée est de [N] bits. Si nous supposons N suffisamment grand, la taille de l'algorithme peut être considérée comme négligeable et donc

$$\alpha_b = \frac{[X]\ \mathrm{Log}_2\ [N]}{[N]}$$

Pour $[X] > [N] / \mathrm{Log}_2\ [N]$ le coefficient α est strictement supérieur à 1.

En fait, il peut même être voisin de $\mathrm{Log}_2\ [N]$ si [X] est voisin de [N].

Exemple :

N est l'ensemble des nombres entiers de 1 à 8 192. X est un sous ensemble de 1 024 éléments.

$$\alpha = \frac{1\ 024 \times 13}{8\ 192} = 1{,}625$$

Le coefficient α_b défini de cette façon n'est donc pas réellement un coefficient d'occupation mémoire, c'est un *coefficient de comparaison* entre la mémoire occupée par la méthode choisie et la mémoire occupée par la méthode séquentielle.

c. Le dernier exemple revenait à effectuer une énumération de l'ensemble des parties de N : $\mathcal{P}(N)$. En effet, cet ensemble possède $2^{[N]}$ éléments. Il suffit donc de $\text{Log}_2\ 2^{[N]}$ bits pour énumérer tous les sous ensembles de N c'est-à-dire [N] bits.

C'est donc le nombre moyen minimal de bits pour représenter n'importe quel sous ensemble de N.

On peut s'intéresser au cas où [X] étant fixé, on désire connaître le nombre moyen de bits nécessaires. De même, on peut faire une énumération dans tous les sous ensembles de N ayant [X] éléments. Il y a $C_{[N]}^{[X]}$ ensembles de cette sorte. Il faut donc $\text{Log}_2\ C_{[N]}^{[X]}$ bits pour les représenter.

Cette formule est peu maniable et peu intéressante en dehors du fait qu'elle est symétrique en [N]/2. Ceci est normal, car représenter un sous ensemble X de N ou son complément permet de répondre de façon identique à la question de savoir si x appartient à X ou non.

Il n'est pas difficile de montrer que $\text{Log}_2 C_{[N]}^{[X]}$ est équivalent à [X] Log [N] / [X] lorsque [N] est grand devant [X] et [X] grand devant zéro.

$$\text{Log}_2\ C_{[N]}^{[X]} \sim [X]\ \text{Log}\ [N]/[X] \qquad \text{si} \quad 0 \ll [X] \ll [N]$$

Cette condition est souvent réalisée dans la pratique et dans les cas défavorables ou [X] est d'un ordre voisin de [N] il suffit d'utiliser la méthode définie dans l'exemple de la partie b.

L'intérêt de la dernière formule est qu'elle est plus maniable, aisément calculable et qu'elle correspond à une idée importante en ce qui concerne le rangement des informations en mémoire.

L'idée est que *l'ordre de rangement des valeurs dans une mémoire* (puisqu'une mémoire est forcément ordonnée) *apporte de l'information* et donc qu'il n'est nécessaire que de ranger le résidu de cette information.

Si nous avons $[X]$ valeurs prises parmi $[N]$, chaque valeur de X dans la mémoire possède un numéro d'ordre qui peut se coder sur $\text{Log}_2[X]$ bits (puisqu'on en a $[X]$) et il suffit donc de ranger $(\text{Log}\,[N] - \text{Log}\,[X])$ bits puisque l'adresse (le numéro d'ordre) apporte l'information manquante.

Comme nous avons $[X]$ valeurs de cette sorte, il suffit donc de $[X](\text{Log}\,[N] - \text{Log}\,[X])$ bits soit $[X]\ \text{Log}\,[N]/[X]$ ce qui était bien la valeur cherchée.

Une meilleure définition de α semble donc être

$$\alpha_c = \frac{[X]\ \text{Log}\,[N]/[X]}{\text{Mémoire utilisée en bits}}$$

Cette définition évite l'obtention d'une valeur α supérieure à 1 dans l'exemple de la partie b.

$$\alpha = \frac{[X]}{[N]}\ \text{Log}\ \frac{[N]}{[X]}$$

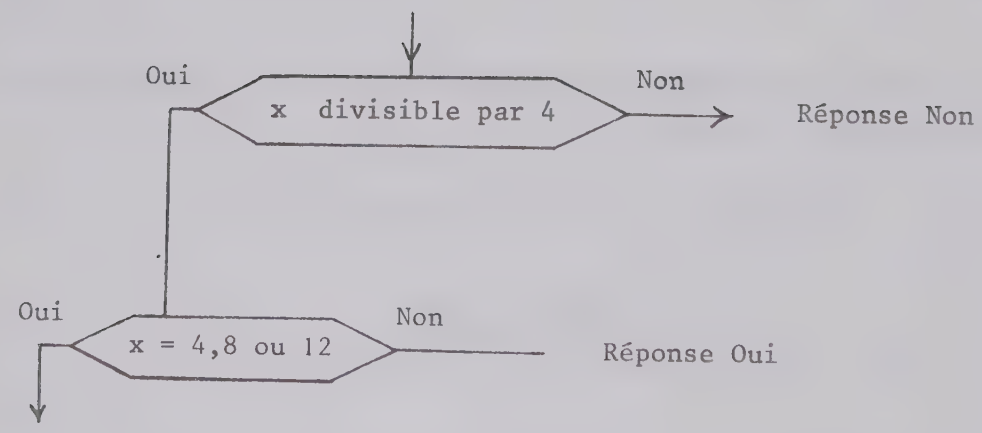
et on a $0 \leqslant \alpha \leqslant 1/2$ dans ce cas.

Remarquons que si X est identique à $[N]$ cette formule $[X]\ \text{Log}\,[N]/[X]$ a quand même un sens. Sa valeur est nulle mais c'est normal, on n'a pas besoin de mémoire pour répondre : oui, x appartient à X.

Nous pouvons dire que ce coefficient α_c est un *coefficient d'occupation de la mémoire relativement à une méthode considérée comme idéale a priori*.

Nous disons "a priori" car il est possible de montrer que ce coefficient α peut encore être supérieur à 1 lorsqu'on connaît (ou qu'on mesure) des informations particulières sur l'ensemble [X] de valeurs à ranger.

Par exemple, si N est l'ensemble des 10 000 premiers nombres entiers et si X est le sous ensemble des nombres divisibles par 4 sauf 4,8 et 12 on peut adopter l'organisation suivante :



Réponse NON

Il suffit de très peu de mémoire (pour les valeurs 4,8 et 12) et pourtant

$$[X] \text{ Log } [N]/[X] \simeq 2\,500 \times 2 = 5\,000 \text{ bits}$$

Mais ce sont des informations "a postériori" qui permettent cette économie de mémoire car elle supposent une connaissance de la structure de l'ensemble X. Notons aussi que la méthode dépend alors essentiellement *de X* et pas seulement de [X] nombre d'éléments de X.

Enfin, l'usage courant étant d'utiliser une coefficient α du type α_b nous utiliserons celui-ci en sachant bien qu'il peut être plus grand que 1 et *qu'avoir $\alpha_b = 1$ ne suffit absolument pas pour dire que la mémoire est pleine !*

2. *Coefficient de recherche : β*

Nous nommerons β le nombre moyen d'accès à la mémoire nécessaire pour retrouver une valeur x dans cette mémoire.

C'est sous cette forme que sont utilisées la plupart des mémoires. On y range un grand nombre de valeurs et on désire retrouver rapidement l'une d'entre elles. Le coefficient β est alors très important car il caractérise souvent directement le temps nécessaire pour effectuer la recherche.

3. *Coefficient de recherche : γ*

Nous nommerons γ le nombre moyen d'accès à la mémoire nécessaire pour montrer qu'une valeur 'y' n'est pas dans la mémoire. Ce problème pourrait se ramener au précédent, puisqu'il suffit de conserver le complémentaire de l'ensemble X dans N et montrer que la valeur 'y' est dans ce complémentaire.

Les ensembles complémentaires sont souvent beaucoup trop considérables et on ne construit pas une mémoire uniquement pour montrer un partage entre β et γ. C'est par exemple, le cas dans les compilateurs où il faut constituer une table des identificateurs définis par l'utilisateur. Ou il faut rechercher des identificateurs dans cette table et de temps en temps, si l'utilisateur se sert d'identificateurs non définis montrer qu'ils ne sont pas dans la table.

Le coefficient γ est en général supérieur à β.

4. *Coefficient d'insertion γ'*

Nous nommerons γ' le nombre moyen d'accès nécessaire pour insérer une nouvelle valeur dans la mémoire. γ' est toujours supérieur à γ si on suppose qu'il faut vérifier que la valeur n'est pas déjà dans la mémoire. Sinon il peut être relativement faible (1 dans la méthode séquentielle).

On peut remarquer qu'introduire une valeur supplémentaire revient à changer le sous ensemble X donc augmenter α_b et par suite, changer le coefficient γ' lui-même.

Ce coefficient est important car pour de petites variations de la taille du fichier et dans le cadre d'une méthode choisie, il donne une mesure de sa possibilité d'évolution. C'est ainsi qu'on trouvera que cette possibilité est lourde et mauvaise avec la méthode dichotomique, puissante dans la méthode par listes, relativement bonne dans le "Hash Coding".

5. *Coefficient de création*

C'est le nombre moyen d'accès nécessaire pour remplir la mémoire jusqu'à un coefficient α déterminé. On peut l'obtenir en intégrant le coefficient γ' entre 0 et α. Nous ne donnons pas de nom à ce coefficient car nous l'emploierons très peu. En effet, on ne crée pas une mémoire pour le besoin de la créer mais pour l'utiliser. On peut très souvent admettre des créations très lourdes en vue d'une utilisation très performante.

6. *Coefficient de retrait* δ

Nous nommerons δ le nombre moyen d'accès nécessaire pour *enlever* une valeur de la mémoire. C'est un coefficient en général élevé car si on désire enlever une valeur, c'est souvent au prix d'une réorganisation partielle de la mémoire, donc au détriment de ce coefficient δ lui-même. On peut encore changer quelque peu la méthode pour que ce coefficient soit acceptable en ajoutant quelques informations supplémentaires mais c'est au détriment du coefficient α et donc des autres coefficients.

C. *CONTRAINTES D'UTILISATION*

Certaines méthodes exigent trois ensembles de valeurs :

N : ensemble des valeurs possibles

X : ensemble des valeurs rangées dans la mémoire

U : ensemble des valeurs d'utilisation

Ce dernier ensemble peut comporter les sous ensembles suivants et peut contenir des répétitions :

U_1 : des valeurs à rechercher

U_2 : des valeurs dont il faut prouver qu'elles ne sont pas dans la mémoire

U_3 : des valeurs à ajouter

U_4 : des valeurs à supprimer

Ce sont les rapports entre le nombre d'éléments de ces divers sous ensembles de valeurs qui importent et qui caractérisent l'utilisation de la mémoire. Soient a_β, a_γ, $a_{\gamma'}$, a_δ les rapports :

$$a_\beta = \frac{[U_1]}{[U]} \qquad a_\gamma = \frac{[U_2]}{[U]} \qquad a_{\gamma'} = \frac{[U_3]}{[U]} \qquad a_\delta = \frac{[U_4]}{[U]}$$

$(a_\beta + a_\gamma + a_{\gamma'} + a_\delta = 1)$

Si on suppose que la taille du fichier X varie faiblement malgré les sous ensembles U_3 et U_4, un bon coefficient de performance de la méthode $\mathcal{M}$ relativement à une utilisation u pour un coefficient α_b (déterminé par la mémoire totale disponible) pourrait être :

$$P(u,\mathcal{M}, \alpha_b) = a_\beta \beta + a_\gamma \gamma + a_{\gamma'} \gamma' + a_\delta \delta$$

Les coefficients β, γ, γ', δ étant fonction de α. Ce coefficient de performance étant en quelque sorte le produit scalaire d'un vecteur caractéristique de la méthode $(\beta, \gamma, \gamma', \delta)$ et d'un vecteur caractéristique de l'utilisation $(a_\beta, a_\gamma, a_{\gamma'}, a_\delta)$.

Remarques

Bien d'autres facteurs peuvent avoir une relative importance sur les caractéristiques, soit de méthodes, soit d'utilisation.

1. Méthode

- *Influence du facteur de groupage*

Dès que les fichiers deviennent assez importants de taille, la mémoire est en fait constituée de mémoires externes (bandes magnétiques, disques, tambours, etc...).

Si les bandes magnétiques conditionnent le mode de recherche, les autres mémoires externes possèdent souvent la particularité de ne permettre l'accès qu'à une quantité fixe d'information qui peut être constituée d'un groupe de valeurs. Le temps d'exécution de l'algorithme étant généralement négligeable devant le temps d'entrée-sorties, il est donc possible de considérer qu'on accède en *un seul accès à un groupe de valeurs*.

On aura donc intérêt dans l'organisation de la mémoire à inclure l'idée de *voisinage* des valeurs dans la mémoire. On ne cherche pas nécessairement à déterminer exactement où se trouve la valeur mais à déterminer un voisinage dans la mémoire où la valeur à trouver possède une bonne chance d'y être.

2. *Utilisation*

- Tous ces coefficients sont des moyennes et de ce fait les variances ont naturellement de l'importance, en particulier parce qu'elles peuvent donner des limites "raisonnables" d'attente dans des problèmes de temps réel ou bien qu'elles peuvent quantifier le risque de ne pas trouver la valeur cherchée dans un temps donné par exemple.

- La fréquence, soit d'apparition, soit de recherche des valeurs, peut aussi être importante. Il peut arriver que la fréquence d'apparition des valeurs (comme résultats d'une expérience en temps réel par exemple) soit grande et de ce fait conditionne toute l'organisation. On ne peut attendre qu'une valeur soit rangée avant de commencer à ranger la suivante. Cette contrainte introduira des suites successives de rangement en cascade au détriment des performances.

- Il peut exister aussi une certaine probabilité d'apparition des valeurs dans la recherche et une bonne méthode devrait tenir compte de ce fait. Les valeurs dont l'apparition est la plus probable doivent être accessibles plus rapidement que les autres.

Sans tenir compte directement de tous ces facteurs, nous essaierons lors de l'étude des divers types d'organisation de déterminer quelques qualités ou défauts relatifs à ces critères.

METHODES D'ORGANISATION

METHODES D'ORGANISATION

Il est possible de définir fondamentalement deux types différents dans les méthodes d'organisation de mémoires :

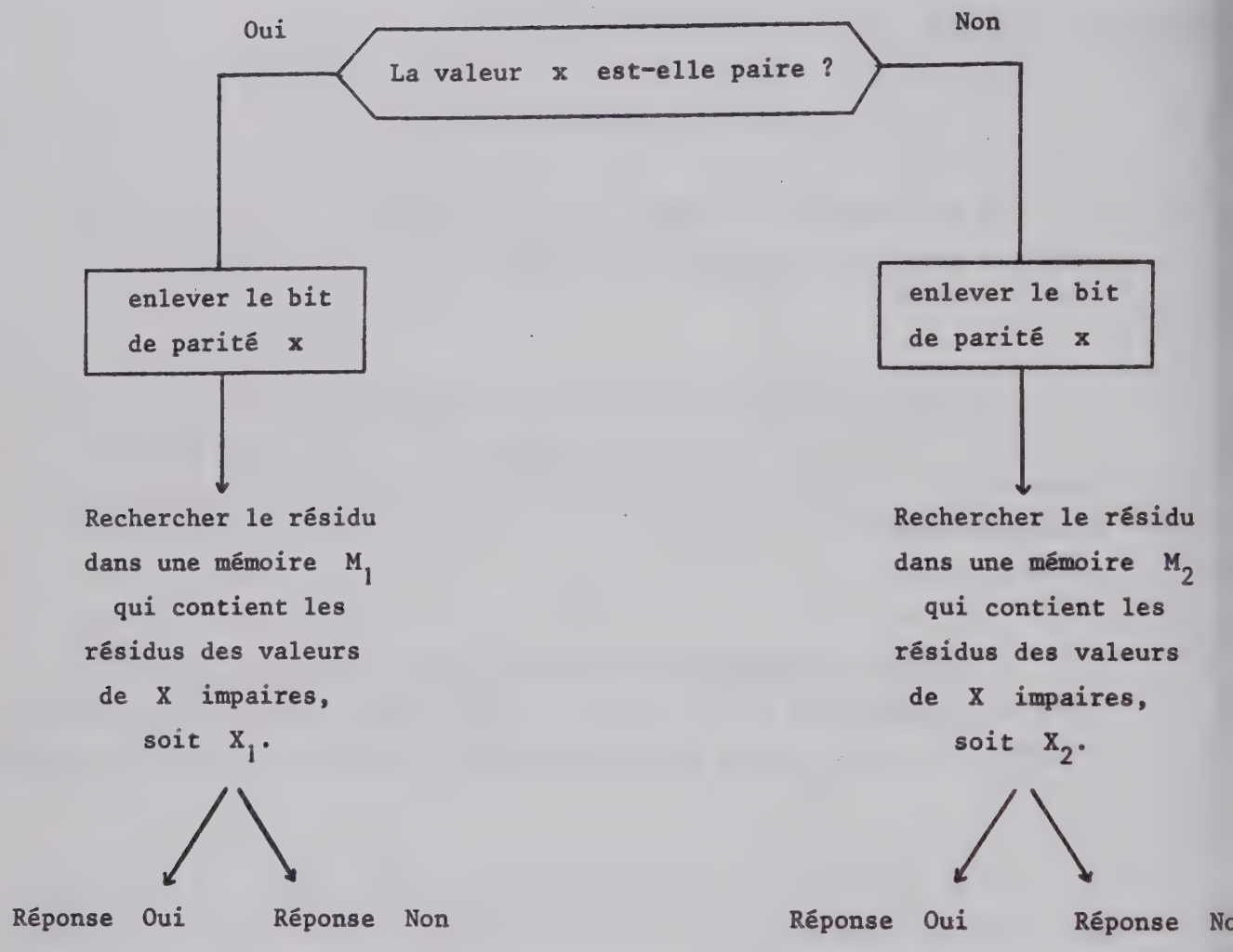
- *L'un "a priori"* lorsque nous ne connaissons rien de spécial sur l'ensemble des valeurs de X par rapport à N (c'est ainsi que nous avons défini le coefficient α_c, coefficient d'occupation relatif à une organisation idéale a priori).

- *L'autre "a postériori"* lorsqu'on suppose (ou qu'on mesure) une certaine connaissance de l'ensemble X qui permet d'effectuer un certain traitement sur les valeurs afin de diminuer la quantité de mémoire utilisée.

Cette deuxième partie pouvant être considérée comme un prétraitement sur l'ensemble des valeurs X avant d'aboutir à une série d'organisations "a priori".

Exemple

Si nous mesurons sur un ensemble de valeurs X qu'il y a deux fois plus de valeurs paires que de valeurs impaires, la méthode d'organisation générale peut être constituée ainsi :



Chacun des deux ensembles X_1 et X_2 peut être conservé par des méthodes d'organisation différentes.

Si X est un ensemble de $[X]$ nombres choisis parmi $[N]$, nous avons vu que si $[X]$ et $[N]$ sont grands, la quantité minimale de mémoire nécessaire "a priori" peut être approchée par

$$[X] \text{ Log } [N]/[X]$$

Après le prétraitement, a postériori, la quantité minimale nécessaire pour chaque ensemble X_1 et X_2 est donc respectivement :

$$[X_1] \text{ Log } \frac{[X_1]}{[N_1]} \quad \text{et} \quad [X_2] \text{ Log } \frac{[X_2]}{[N_2]}$$

N_1 et N_2 sont les ensembles constitués des valeurs résidus lorsqu'on envoie *toutes* les valeurs de N dans le prétraitement.

Ces quantités peuvent s'écrire dans notre exemple :

$$\frac{2[X]}{3} \text{ Log } \frac{[N] \times 3}{2[X] \times 2} \quad \text{et} \quad \frac{[X]}{3} \text{ Log } \frac{[N] \times 3}{2 \times [X]}$$

Soit au total :

$$\frac{2[X]}{3} \text{ Log } \frac{3}{4}\frac{[N]}{[X]} + \frac{[X]}{3} \text{ Log } \frac{3}{2}\frac{[N]}{[X]}$$

Soit $$[X] \text{ Log } \frac{[N]}{[X]} + \frac{2[X]}{3} \text{ Log } \frac{3}{4} + \frac{[X]}{3} \text{ Log } \frac{3}{2}$$

Nous avons donc un gain "théorique" de mémoire dû au prétraitement de $[X]$ $(\frac{2}{3} \text{ Log } \frac{3}{4} + \frac{1}{3} \text{ Log } \frac{3}{2})$ soit $0,1[X]$ approximativement.

On peut donc dire que le prétraitement permet un gain de mémoire de $0,1[X]$ connaissant X ou encore qu'il contient une information évaluée à $0,1[X]$.

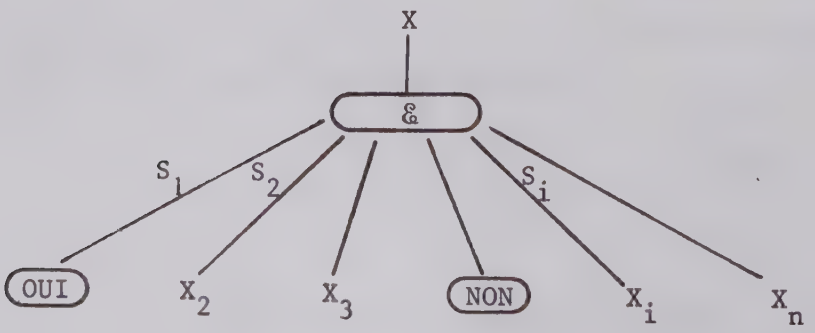
A. *PRETRAITEMENT*

1. *Mesure du gain de mémoire*

Nous considérerons que le prétraitement est relativement réduit et que la quantité de mémoire nécessaire pour sa représentation reste négligeable devant le reste de la mémoire utilisée.

Soit & un algorithme de prétraitement sur un ensemble de valeurs X prises parmi N.

Seules importent dans ce prétraitement les diverses sorties s_i dont l'ensemble est S.



Ces sorties aboutissent soit à des réponses certaines (oui ou non), soit à des sous mémoires X_i. Supposons que le prétraitement sépare a priori l'ensemble des N valeurs possibles en des ensembles de $[N_1]$, $[N_2]$, $[N_n]$ valeurs chacun.

Nous définissons

$$C_i = \frac{[N_i]}{[N]} , \quad \forall i,$$

avec $$\sum_{i=1}^{[N]} C_i = 1 \quad \text{et} \quad C_i \neq 0 \ \forall i$$

Supposons aussi que le prétraitement sépare l'ensemble des $[X]$ valeurs (le volume initial de mémoire) en des ensembles de $[X_1]$, $[X_2]$, $[X_n]$ valeurs.

Nous définissons de même

$$P_i = \frac{[X_i]}{[X]} \quad \text{avec} \quad \sum_{i=1}^{n} P_i = 1, \quad \forall i$$

Certains p_i peuvent être nuls, ils correspondent à des branches "non". En dehors des branches à sortie certaine "oui", nous supposerons dans la suite

$$0 \ll [X_i] \ll \frac{[N_i]}{2} , \quad \forall i \quad \text{et} \quad 0 \ll [X] \ll \frac{[N]}{2}$$

Chaque mémoire M_i est composée de $p_i[X]$ valeurs prises parmi $C_i[N]$ possibles. La quantité moyenne minimale a priori de mémoire pour conserver ces valeurs est pour chaque mémoire $[X_i]$ Log $([N_i]/[X_i])$ bits. Soit :

$$[X]\ P_i\ \text{Log}\ \frac{C_i}{P_i} + P_i[X]\ \text{Log}\ \frac{[N]}{[X]}$$

et la mémoire nécessaire pour les n branches du prétraitement

$$\sum_{i=1}^{n} \left([X]\ P_i\ \text{Log}\ \frac{C_i}{P_i} + P_i[X]\ \text{Log}\ \frac{[N]}{[X]}\right)$$

Soit $$[X] \sum_{i=1}^{n} P_i\ \text{Log}\ \frac{C_i}{P_i} + [X]\ \text{Log}\ \frac{[N]}{[X]}$$

et le gain est donc

$$g = [X] \sum_{i=1}^{n} P_i\ \text{Log}\ \frac{P_i}{C_i}$$

Exemple

1. Reprenons le dernier exemple avec la question : la valeur de x est-elle paire ou impaire ?

   On a donc deux ensembles X_1 et X_2.

   Réponse paire X_1 : $C_1 = 1/2$ $P_1 = 2/3$

   Réponse impaire X_2 : $C_2 = 1/2$ $P_2 = 1/3$

   et donc

$$g = [X] \left(\frac{2}{3} \text{Log} \frac{4}{3} + \frac{1}{3} \text{Log} \frac{2}{3}\right)$$

conformément à ce qui avait été trouvé.

2. Prenons comme exemple des chaînes de caractères dont on sait que le dernier est toujours A.

   Supposons la codification des caractères sur 6 bits.

   Avec la question : le dernier caractère est-il A ? on a

X_1 : $C_1 = 1/64$ $P_1 = 1$

X_2 : $C_2 = 63/64$ $P_2 = 0$

   et donc

$$g = [X] \text{ Log } 64 = 6 [X] \text{ bits}$$

ce qui est bien prévisible, il suffit de tronquer le caractère A à toutes les chaînes.

3. Considérons toujours des chaînes mais tous les caractères étant alphabétiques. Supposons toujours une codification des caractères sur 6 bits.

Avec la question : le $j^{ème}$ caractère est-il alphabétique ? (lettre ou blanc) on a

$$X_{i1} : \quad C_{i1} = \frac{27}{64} \qquad P_{i1} = 1$$

$$X_{i2} : \quad C_{i2} = \frac{37}{64} \qquad P_{i2} = 0$$

soit un gain $g = [X]\ \text{Log}\ (64/27)$ et si on suppose toutes les chaînes de 5 lettres

$$g = 5\ [X]\ \text{Log}\ (64/27)$$

ce qui est bien naturel car il suffit d'effectuer un codage à base 27 au lieu de 64 (6 bits).

2. *Propriétés du gain*

Propriété 1 : *le gain est positif ou nul*

Cette propriété est triviale lorsqu'on effectue un rapprochement entre le gain et l'information mutuelle (voir plus loin) mais certaines inégalités intervenant dans la démonstration étant utiles par la suite, nous préfèrons en donner une démonstration directe.

Soit donc à démontrer

$$(1) \qquad g = [X] \sum_{i=1}^{n} P_i \operatorname{Log} \frac{P_i}{C_i} \geqslant 0$$

avec les hypothèses suivantes $\begin{cases} \sum P_i = \sum C_i = 1 \\ \forall i \quad P_i \geqslant 0 \\ \forall i \quad C_i > 0 \end{cases}$

- Nous commencerons par montrer l'inégalité suivante

$$(2) \qquad (P_1 + P_2) \operatorname{Log} \frac{P_1 + P_2}{C_1 + C_2} \leqslant P_1 \operatorname{Log} \frac{P_1}{C_1} + P_2 \operatorname{Log} \frac{P_2}{C_2}$$

avec les hypothèses suivantes $\begin{cases} P_1 + P_2 \leqslant 1 \quad P_1, P_2 \geqslant 0 \\ C_1 + C_2 \leqslant 1 \quad C_1, C_2 > 0 \end{cases}$

Il suffit d'évaluer

$$F = P_1 \operatorname{Log} \frac{P_1 + P_2}{C_1 + C_2} + P_2 \operatorname{Log} \frac{P_1 + P_2}{C_1 + C_2} - P_1 \operatorname{Log} \frac{P_1}{C_1} - P_2 \operatorname{Log} \frac{P_2}{C_2}$$

Posons $\frac{P_2}{C_2} = \frac{P_1}{C_1} + k$ et supposons $k \geqslant 0$ ce qui ne restreint pas le problème.

Alors $F = P_1 \text{ Log } (1 + \frac{k\, C_1\, C_2}{P_1 (C_1 + C_2)}) + P_2 \text{ Log } (1 - \frac{k_1\, C_1\, C_2}{P_2 (C_1 + C_2)})$

en posant $k' = \frac{k\, C_1\, C_2}{C_1 + C_2}$ on vérifie facilement que k' est compris entre 0 et P_2 et que la fonction $F(x) = P_1 \text{ Log } (1 + \frac{x}{P_1}) + P_2 \text{ Log } (1 - \frac{x}{P_2})$ est négative entre 0 et P_2 donc pour la valeur de k'. Ceci démontre notre inégalité (2).

- Montrons maintenant

$$(3) \qquad (\sum_{i=1}^{n} P_i) \text{ Log } \frac{(\sum_{i=1}^{n} P_i)}{(\sum_{i=1}^{n} C_i)} \leqslant \sum_{i=1}^{n} P_i \text{ Log } \frac{P_i}{C_i}$$

avec les hypothèses $\begin{cases} \sum P_i \leqslant 1, \ \forall i, & P_i \geqslant 0 \\ \sum C_i \leqslant 1, \ \forall i, & C_i > 0 \end{cases}$

On a d'après l'inégalité (2)

$$(\sum P_i) \text{ Log } \frac{(\sum P_i)}{(\sum C_i)} \leqslant P_1 \text{ Log } \frac{P_1}{C_1} + (\sum_{i \neq 1} P_1) \text{ Log } \frac{(\sum_{i \neq 1} P_i)}{(\sum_{i \neq 1} C_i)}$$

et par inductions successives on démontre ainsi l'inégalité (3).

Si on pose $\sum P_i = \sum C_i = 1$ dans (2) on démontre alors l'inégalité (1).

Sorties certaines

1. Sorties certaines pour l'ensemble des valeurs de N.

Ce sont aussi des sorties certaines pour les valeurs de X. C'est donc un cas trivial.

- si la sortie est NON, la sortie est complètement neutre, on n'y passe jamais.

- si la sortie est OUI, c'est le reste du prétraitement qui est neutre, on passera toujours par cette sortie.

Ces deux cas peuvent être éliminés, c'est pourquoi nous avons supposé $C_i > 0$ (strictement). On peut donc aussi supposer $C_i < 1$.

2. Sorties certaines pour les valeurs de X.

a. sorties NON

Il ne passe aucune valeur de X par la sortie. On a donc :

$$C_i \neq 0 \qquad P_i = 0$$

l'inégalité (2) nous donne alors

$$\sum_{j \neq i} P_i \operatorname{Log} \frac{P_j}{C_j} \geqslant \left(\sum_{j \neq i} P_i \right) \operatorname{Log} \frac{\left(\sum_{i \neq j} P_j \right)}{\left(\sum_{j \neq i} C_i \right)}$$

mais comme $P_i = 0$

$$\sum_{i \neq j} P_j \text{ Log } \frac{P_j}{C_j} = \sum_j P_j \text{ Log } \frac{P_j}{C_j}$$

et $$\sum_{j \neq i} P_j = 1 \qquad \sum_{j \neq i} C_j = 1 - C_i \neq 1$$

et donc

$$g = [X] \sum_j P_j \text{ Log } \frac{P_i}{P_j} = [X] \sum_{j \neq i} P_j \text{ Log } \frac{P_j}{C_j} \geqslant [X] \text{ Log } \frac{1}{1-C_j} > 0$$

On peut donc énoncer

Propriété 2 : Pour une sortie certaine NON, le gain est majoré par la quantité $[X]$ *Log* $(1/(1 - C_j))$

b. sorties OUI

Une sortie OUI sur les valeurs de $[X]$, passant par une branche j signifie que toutes les valeurs de N qui passent par cette branche appartiennent aussi à X.

Donc $$C_i [N] = P_i [X]$$

La contribution au gain apportée par la sortie i est donc

$$[X] P_i \text{ Log } \frac{[N]}{[X]}$$

Cette contribution est donc strictement positive et linéairement fonction de P_i.

3. *Relation avec les grammaires*

Une grammaire peut être considérée comme la description d'un automate permettant de déterminer si un mot appartenant à un monoïde T^* (notre ensemble N) appartient ou n'appartient pas à un sous ensemble L (notre ensemble X) de ce monoïde nommé langage.

Si le langage est fini, il est possible de ranger simplement les éléments de L dans une mémoire et d'effectuer une recherche dans cette mémoire pour reconnaître un élément. La grammaire n'est autre qu'une description de la structure du langage permettant de construire un algorithme possédant deux sorties Oui et Non et n'utilisant aucune mémoire.
On remplace en fait complètement la mémoire par un prétraitement.

- Le problème vient de ce que l'ensemble T^* n'est certainement pas fini et que l'ensemble L peut ne pas l'être non plus.

Par contre, les quantités C_i et P_i, définies comme étant des fréquences de "passage" des valeurs dans les différentes sorties du prétraitement, sont des quantités finies (< 1). On peut donc quand même donner une mesure "relative" du gain qui serait :

$$g_{relatif} = \sum_{i=1}^{n} P_i \operatorname{Log} \frac{P_i}{C_i}$$

Une grammaire ne possède en fait que des branches à sortie "Oui" (x appartient au langage L) ou des branches à sortie "Non" (x n'appartient pas au langage L) on a donc :

$$g_r = \sum_{\substack{\text{branches à} \\ \text{sorties Non}}} P_i \operatorname{Log} \frac{P_i}{C_i} + \sum_{\substack{\text{branches à} \\ \text{sorties Oui}}} P_i \operatorname{Log} \frac{P_i}{C_i}$$

$$= 0 + \sum_{\text{sorties Oui}} P_i \operatorname{Log} \frac{P_i}{C_i}$$

Or $\frac{P_i}{C_i} = d$ densité du langage dans le monoïde car pour ces branches $C_i [N] = P_i [X]$ et comme $\sum_{\text{sorties Oui}} P_i = \sum_{i=1}^{n} P_i$

on a

$$g_r = - \text{Log}_2 \, d$$

Le gain relatif est égal à l'opposé du logarithme de la densité du langage dans le monoïde.

- Dans le cas où le langage est fini, on peut définir un gain réel

$$g = [X] \sum_{i=1}^{n} P_i \, \text{Log} \, \frac{P_i}{C_i}$$

mais pour les sorties "Oui" on a nécessairement $C_i = 0$ ce qui rend le gain infini.

Ceci n'est pas naturel, car il semble qu'on puisse ranger le langage L entier dans une mémoire finie et donc que le gain apporté par la structure ne puisse être que fini.

Il est possible de donner une solution convenable en changeant la définition des coefficients C_i. En fait, pour définir C_i on avait supposé que l'ensemble N (ou T^*) se partageant en N_1, N_2, ..., N_n sous ensembles. Certains peuvent maintenant être de cardinal fini.

On avait défini $C_i = \frac{[N_i]}{[N]}$ ce qui ne posait pas de problème dans le cas fini, mais peut donner un rapport égal à 0 si $[N]$ est fini et $[N]$ infini.

Définissons $\overline{N} = \sum_{[N_i] \neq \infty} [N_i]$ et $C_i = \frac{[N_i]}{\overline{N}} \quad \forall i$

le gain est alors égal à $g = [X] \sum_{\text{sorties Oui}} P_i \text{ Log } \frac{P_i}{C_i}$

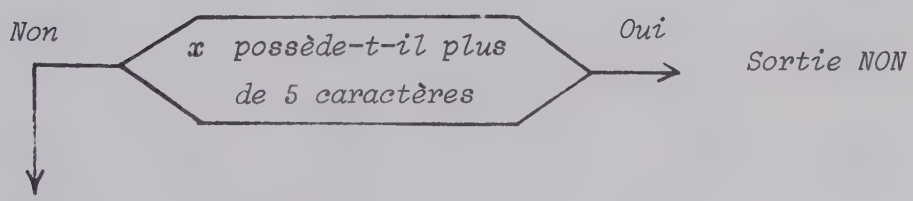
on a toujours $C_i \overline{N} = P_i [X]$ et donc

$$g = [X] \text{ Log } \frac{\overline{N}}{[X]}$$

Cette définition du gain aboutit malgré tout à une idée assez naturelle dans le cas d'un prétraitement d'un ensemble fini contenu dans un ensemble infini.

Exemple

Supposons le prétraitement composé d'une question à sortie NON infinie et d'une sortie aboutissant à un algorithme qui puisse se décrire par une grammaire, le nombre d'éléments de N passant par cette branche étant fini.



Algorithme de reconnaissance

La sortie NON de la première question est la seule de ce type ($[N_i] = \infty$) et on peut considérer que puisque x peut toujours se coder sur 5, caractères, $\overline{N}$ est l'ensemble des mots de 5 caractères au plus.

La quantité minimale moyenne a priori pour conserver le langage X est donc :

$$[X] \ \mathrm{Log} \ \frac{\overline{N}}{[X]}$$

C'est aussi le gain qu'apporte le prétraitement associé à la grammaire de la sortie Non (on gagne tout le langage) et aussi le gain obtenu par le prétraitement global.

4. Relation avec l'information mutuelle

Les éléments de N sont les seules réalisations d'évènements possibles, les différents C_i peuvent donc être considérés comme les probabilités des éventualités du prétraitement s_i,

$$C_i = p(s_i)$$

Les p_i peuvent être considérés comme les probabilités des éventualités du prétraitement lorsqu'on sait que la valeur X est un élément de X,

$$P_i = P(s_i \mid x \in X) \quad \text{que nous notons} \quad P(s_i \mid X)$$

L'expression donnant le gain peut donc s'écrire :

$$g = [X] \sum_{i=1}^{n} P(s_i \mid X) \ \mathrm{Log} \ \frac{P(s_i \mid X)}{P(s_i)}$$

la quantité $\sum_{i=1}^{n} P(s_i \mid X) \ \mathrm{Log} \ \frac{P(s_i \mid X)}{P(s_i)}$ n'est autre

que "l'information mutuelle" entre S ensemble des sorties s_i du prétraitement et X ensemble des valeurs sur lesquelles ce prétraitement opère.

$$g = [X] \cdot I[S ; X]$$

La propriété (1) s'en déduit directement, on peut même annoncer que dès qu'un coefficient P_i est différent d'un coefficient C_i correspondant, le gain est strictement positif.

Il est possible d'énoncer une troisième propriété :

Propriété 3 : le gain de mémoire réalisé par un prétraitement sur un ensemble de valeurs X est égal au produit du nombre d'éléments de X par l'information mutuelle entre le prétraitement et l'ensemble X.

La notion d'information mutuelle prend ainsi un sens très concret, lié à l'économie de volume de mémoire nécessaire pour ranger une représentation.

Ceci n'est valable bien sûr que si N est suffisamment grand mais aussi si X et N - X sont grands. Ces deux dernières restrictions semblent importantes, elles proviennent du fait qu'on a appliqué la formule de stirling pour réaliser une approximation de [X] ! et [N - X] ! Les valeurs de [X] et [N - X] n'ont cependant pas besoin d'être énormes.

Pour de petites valeurs de [X] ou [N - X] il suffit de penser que pour une valeur de N suffisamment grande, les branches s_i, aboutissant à des ensembles X_i ou $N_i - X_i$ de cet ordre, sont très peu différentes de sorties OUI ou NON. Leur contribution en mémoire est faible devant la mémoire initiale.

La formule réelle du gain, sans approximations serait la suivante :

$$g = \text{Log} \frac{[N]\ !}{\pi\ [N_i]\ !} - \text{Log} \frac{[X]\ !}{\pi\ [X_i]\ !} - \text{Log} \frac{[N - X]\ !}{\pi\ [N_i - X_i]\ !}$$

Elle possède l'avantage d'être symétrique en X_i et $N_i - X_i$ mais ne permet pas beaucoup de développements intéressants. Elle n'est pas aisément calculable, elle se réduit à $[N]$ calculs de logarithmes puis $[N] + 3n + 6$ additions ce qui n'est pas considérable mais les logarithmes étant déjà des approximations et étant donnée la forme de l'expression (différence de deux quantités très importantes), le calcul risque de ne rien donner de cohérent sur un ordinateur.

5. *Construction d'un prétraitement*

Nous avons aboutit dans les chapitres précédents à une mesure qui semble valable puisqu'elle donne des résultats correspondant aux résultats intuitifs, de la qualité d'un prétraitement.

On peut alors se poser le problème suivant :

Problème : *construire un prétraitement intéressant sur un ensemble de données X.*

a) On désire que le prétraitement n'ait que des sorties OUI ou NON.

Si l'ensemble de données X est considéré comme un langage, cela revient à construire une grammaire permettant d'engendrer le langage. C'est un problème bien connu sous le nom d'inférence grammaticale. Ce sujet a déjà été très étudié, en particulier par J. FELDMAN qui s'est vite aperçu que ses résultats pouvaient s'étendre à autre chose que des grammaires et qu'en fait, l'inférence grammaticale n'est qu'un cas particulier de ce qu'on pourrait appeler inférence d'algorithme.

La principale difficulté dans le passage de l'inférence grammaticale à l'inférence d'algorithme provient du fait que la notion de grammaire est bien formalisée et qu'il est possible de construire assez facilement une énumération de l'ensemble des grammaires. Par contre, la notion d'algorithme est encore contreversée. La formalisation la plus intéressante à notre avis semble être celle de ZOHAR MANA.

L'inconvénient majeur des méthodes d'inférence est qu'elles nécessitent une énumération de l'ensemble des algorithmes. On doit généralement parcourir cet ensemble suivant cette énumération en essayant chaque algorithme jusqu'à ce qu'on en trouve un d'intéressant.

b) On laisse les sorties du prétraitement aboutir à des mémoires partielles.

Nous donnerons dans cette partie une méthode de construction d'un prétraitement en utilisant une notion d'opérateurs proche de celle de ROCHE étendue avec le formalisme décrit par ZOHAR MANA.

- *Définition des opérateurs*

Nous définirons un opérateur θ sur un domaine D non vide comme étant *un ensemble fini de prédicats* $\{P_i(x)\}_{i \in [1n]}$ chaque prédicat étant associé à une *fonction* $f_i(x)$ de D dans D appelée *réduction associée*.

L'ensemble de prédicats possède les propriétés suivantes :

- il est complet

$$\forall x \quad (P_1(x) \vee P_2(x) \ldots \vee P_n(x))$$

- il est mutuellement exclusif

$$\forall x \quad (\sim P_i(x) \vee \sim P_j(x))$$

Pour toute paire (i, j) $\qquad 1 \leqslant i \neq j \leqslant n$

Ces conditions signifient que pour un x donné il n'y a qu'un seul prédicat qui prend la valeur vraie.

L'ensemble des fonctions de réduction réduit l'information x au sens où nous l'avons vu dans les chapitres précédents. Si l'information x se présente sous une certaine représentation dans D, (D est l'ensemble des représentations possibles) le prédicat ayant une valeur, la transformation associée fournit le résidu obtenu en enlevant à x cette information. Nous supposerons que ces fonctions $f_i(x)$ existent toujours pour les opérateurs que nous considérons.

Etant donnés deux sous ensembles de D : D_N et D_X tels que $D_X \subset D_N$ on peut toujours définir, comme dans le chapitre (III-A-1), le gain de l'opérateur θ relativement à D_N et D_X.

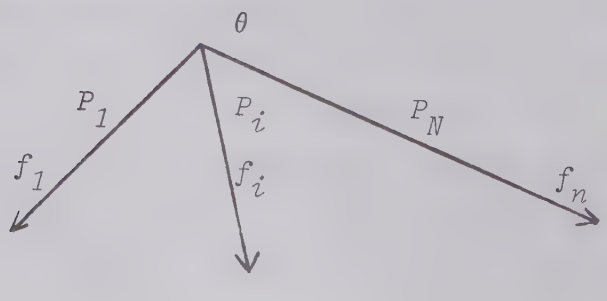
L'opérateur divise le domaine D_N en fournissant comme sorties des fonctions f_i (qui ne s'appliquent que lorsque le prédicat est vrai) des sous ensembles de D : $D_{N_1}, D_{N_2}, \ldots, D_{N_n}$. De même, les fonctions f_i fournissent des sous ensembles $D_{X_1}, D_{X_2}, \ldots, D_{X_n}$ en définissant

$$P_i = \frac{[D_{X_i}]}{[D_X]}, \quad \forall\, i$$

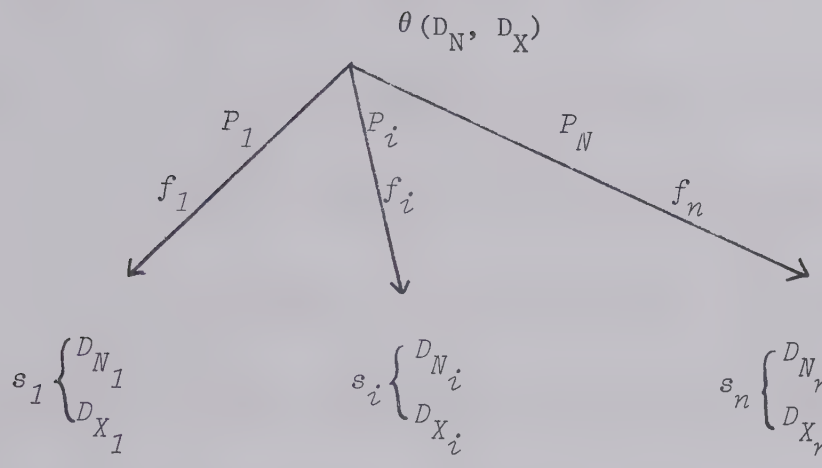
$$C_i = \frac{[D_{N_i}]}{[D_N]}, \quad \forall\, i$$

$$g_{\theta, D_N, D_X} = D_X \sum_{i=1}^{n} P_i \operatorname{Log} \frac{P_i}{C_i}$$

Il est possible de représenter graphiquement un opérateur par



et sa réalisation sur D_N et D_X



Lorsque deux sous ensembles D_{N_i} et D_{X_i} sont identiques, nous disons que la sortie s_i de l'opérateur est OUI, si D_{X_i} est vide (D_{N_i} non vide) nous disons que la sortie s_i est NON.

- *Composition d'opérateurs*

Etant donné un opérateur θ_1, caractérisé par $\{P_i(x)\}_{i \in [1, n]}$ ensemble de prédicats et $\{f(x)\}$ réductions associées, et un opérateur θ_2 caractérisé par $\{P'_j(x)\}_{j \in [1\ k]}$ et $\{f_j(x)\}$.

Nous définirons une loi de composition suivante $1 \in [1\ n]$ notée $\theta_1 \overset{1}{} \theta_2$ entre les opérateurs θ_1 et θ_2 par l'opérateur θ' défini par ses ensembles de prédicats $\{q_r(x)\}_{r \in [1\ t]}$ et de réductions associées $\{g_r'x)\}$:

$$\theta' \begin{cases} \{q_r(x)\}_{r \in [1\ t]} = \{P_i(x)\}_{i \neq 1} \cup \{P'_j(f_1(x)) \wedge P_1(x)\} \\ \{g_r(x)\}_{r \in [1\ t]} = \{g_i(x)\}_{i \neq 1} \cup \{f'_j(f_1(x))\}_{j \in [1, k]} \end{cases}$$

avec $t = n + k - 1$

On vérifie aisément que θ' est bien un opérateur

- $\{q_r(x)\}$ est complet

(1) $$\forall x \ (q_1(x) \vee q_2(x) \ldots \vee q_t(x))$$

est identique à

(2) $$\forall x \ (P_1(x) \vee P_2(x) \ldots \vee P_{l-1}(x) \vee P_{l+1}(x) \ldots \vee P_n(x)$$

$$\vee (P'_1(f_1(x)) \wedge P_1(x)) \vee \ldots \vee (P'_j(f_1(x)) \wedge P_1(x)) \vee \ldots P'_k(f_1(x)) \wedge P_1(x)$$

La deuxième partie de l'expression (2) se réduit à $[P'_1(f_1(x)) \vee P'_2(f_1(x)) \ldots \vee P'_k(f_1(x)) \wedge P_1(x)]$ et donc l'expression (2) devient :

(3) $$\forall x \ ((P_1(x) \vee P_2(x) \ldots \vee P_{l-1}(x) \vee P_1(x) \vee P_{l+1}(x) \ldots \vee P_n(x)$$

$$(P'_1(f_1(x)) \vee P'_2(f_1(x)) \ldots \vee P'_k(f_1(x)) \vee K)$$

Du fait que θ_1 et θ_2 sont des opérateurs on en déduit alors que $\{q_r\}$ est complet.

- $\{q_r(x)\}$ est mutuellement exclusif

$$\forall x \ (\sim q_r(x) \vee \sim q_s(x))$$

s'écrit :

- si q_r et q_s proviennent de $\{P_i(x)\}_{i \neq 1}$

$\forall x \ (\sim P_{r_1}(x) \vee \sim P_{s_1}(x))$ or θ_1 est un opérateur

- si q_r et q_s proviennent de l'un et l'autre des sous ensembles de prédicats

$$\forall x \ (\sim P_{r_1}(x) \ V \sim P'_{s_1}(f_1(x)) \ V \sim P_1(x))$$

or on a du fait que θ_1 est un opérateur $\forall x \ (\sim P_{r_1}(x) \ V \sim P_1(x))$

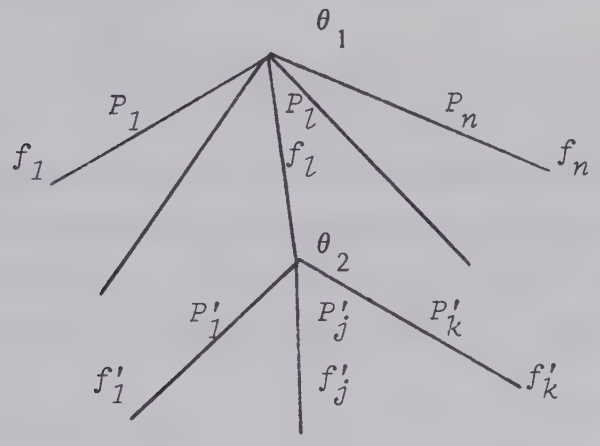
- si q_r et q_s proviennent de $\{P'_j(f_1(x) \sim P_1(x))\}$

$$\forall x \ (\sim P'_{r_i}(f_1(x)) \ V \sim P_1(x) \ V \ P'_{s_1}(f_1(x)) \ V \sim P_1(x))$$

or on a $\forall x \ (\sim P'_{r_1}(f_1(x)) \ V \ P'_{s_1}(f_1(x)))$ du fait que θ_2 est un opérateur.

Les réductions $g_r(x)$ étant des compositions directes de deux fonctions de réductions elles ne peuvent que satisfaire notre condition.

On peut représenter graphiquement l'opérateur θ' résultant de $\theta_1^1 \ \theta_2$ ainsi :



Nous considérerons que la loi de composition n'est possible (sinon elle n'est pas intéressante) que si s_1 n'est pas une sortie OUI ou NON.

- Gain résultant d'une composition d'opérateurs

On a $$g_{\theta', D_N, D_X} = [D_X] \sum_{r=1}^{t} P_r^N \text{ Log } \frac{P''_r}{C''_r}$$

$$= [D_X] \sum_{\substack{\text{sorties directes} \\ \text{de } \theta_1}} P''_r \text{ Log } \frac{P''_r}{C''_r} + [D_X] \sum_{\text{autres sorties}} P''_r \text{ Log } \frac{P''_r}{C''_r} \qquad (1)$$

Le premier terme de (1) est identique à

$$[D_X] \sum_{i \neq 1} P_i \text{ Log } \frac{P_i}{C_i} = g_{\theta_1, D_N, D_X} - [D_X] P_1 \text{ Log } \frac{P_1}{C_1} \qquad (2)$$

puisque sur ces sorties, les coefficients P_i et C_i sont identiques aux coefficients P''_r et C''_r correspondants.

Pour le deuxième terme, on a

$$P''_r = \frac{[D_{X_r}]}{[D_X]}$$

D_{X_r} est l'ensemble engendré par $f'_r(f_1(x))$ lorsque le prédicat $P'_r(f_1(x)) \wedge P_1(x)$ est vrai ou encore $f'_r(D_{X_1})$ lorsque $P'_r(f_1(x))$ est vrai

P''_r peut s'écrire $\frac{[D_{X_r}]}{[D_{X_1}]} \times \frac{[D_{X_1}]}{[D_X]}$ soit $p_1 \times P'_r$.

Les coefficients P'_r étant définis comme précédemment en considérant que l'opérateur θ_2 s'applique avec les domaines D_{N_1} et D_{X_1}.

On déduit de même $C''_r = C_1 \times C'_r$ et donc le deuxième terme de (1) s'écrit

$$[D_X] \sum_{j=1}^{k} P_1 \times P'_j \text{ Log } \frac{P_1 \, P'_j}{C_1 \, C'_j}$$

soit $[D_X] \sum_{j=1}^{k} P_1 P'_j \text{ Log } \frac{P_1}{C_1} + [D_X] \sum_{j=1}^{k} P_1 P'_j \text{ Log } \frac{P'_j}{C'_j}$

$= (D_X \sum_{j=1}^{k} P'_j) P_1 \text{ Log } \frac{P_1}{C_1} + [D_X] P_1 \sum_{j=1}^{k} P'_j \text{ Log } \frac{P'_j}{C'_j}$

on a $\sum_{j=1}^{k} P'_j = 1$ et $[D_X] P_1 = [D_X] \times \frac{[D_{X_1}]}{[D_X]} = D_{X_1}$ en utilisant (2)

on trouve

$$g_{\theta'}, D_N, D_X = g_{\theta_1}, D_N, D_X + g_{\theta_2}, D_{N_1}, D_{X_1}$$

- *Algorithme de construction d'un prétraitement*

Nous supposons un ensemble V fini d'opérateurs dits opérateurs élémentaires. Nous venons de voir qu'il est possible de combiner ces opérateurs pour en former d'autres.

Le nombre de combinaisons que l'on peut faire à partir d'un opérateur quelconque est fini. En effet, le nombre de sous ensembles D_{X_i} non vides est croissant au fur et à mesure que l'on compose les opérateurs (car un opérateur ne peut pas comporter que des sorties NON) or ce nombre est fini puisque l'ensemble D_X de départ est fini par hypothèse.

Soit K le nombre de sous ensembles D_{X_i} maximum. Deux cas peuvent se produire :

- pour chacun des D_{X_i} on a $D_{X_i} = D_{N_i}$. C'est une sortie Oui et il n'est pas possible de combiner un autre opérateur à celui obtenu.

- il existe un $D_{X_i} \neq D_{N_i}$ alors il est possible d'appliquer n'importe quel opérateur élémentaire à cette sortie. Cet opérateur ne peut

avoir que des sorties NON et *une sortie* donnant un D_{X_1} sinon le nombre K augmenterait. Si cette sortie est une sortie OUI on a éliminé une des sorties $D_{X_i} \neq D_{N_i}$. Si cette sortie n'est pas une sortie OUI il est possible de recommencer à appliquer n'importe quel opérateur. Ce procédé ne peut que s'arrêter car les fonctions de réductions employées successivement sont strictes et comme l'information de départ est finie, on ne peut arriver qu'à un ensemble vide ou une sortie OUI.

Un algorithme de construction d'un prétraitement peut être le suivant :

1. On commence par choisir l'opérateur élémentaire θ_1 tel que $g_{\theta 1}$, D_N, D_X soit maximum.

2. Choisir à nouveau parmi les opérateurs élémentaires, les opérateurs θ'_1, tel que $g_{\theta^i_1}$, θ'_{1i}, D_N, D_X soit maximum pour chaque i.

3. Choisir parmi les opérateurs θ'_{1i} celui qui donne le meilleur gain. Soit θ_2 l'opérateur produit $\theta^i_1 \theta'_{1i}$.

On recommence ensuite de même à partir de l'étape 2 avec l'opérateur θ_2.

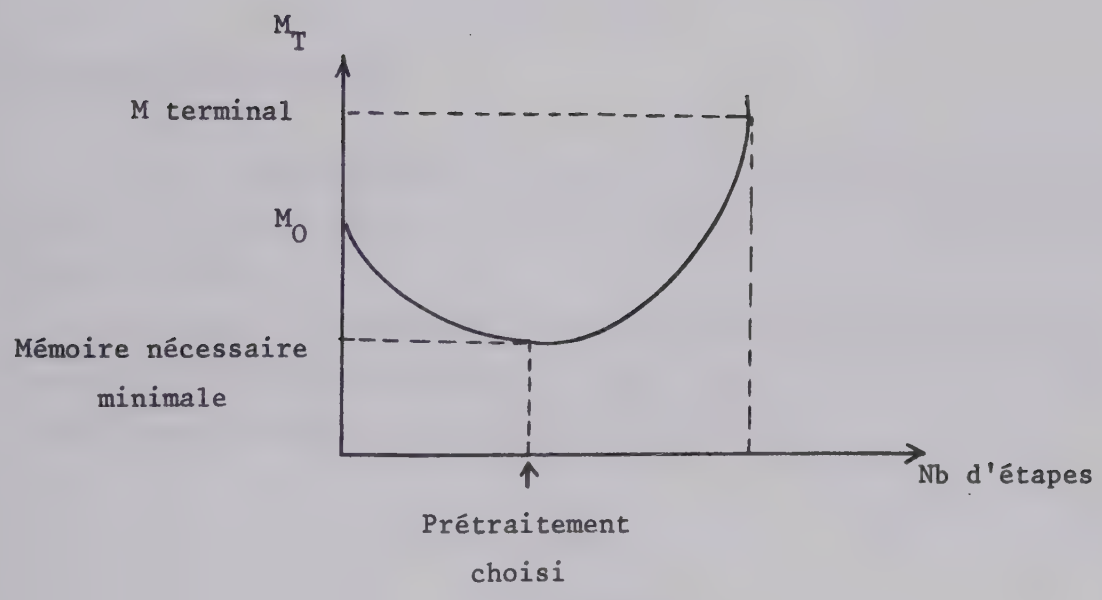
Nous avons vu que cet algorithme s'arrête nécessairement. Il est efficace car à chaque étape, nous choisissons l'opérateur qui apporte le meilleur gain. Seulement si on poussait le procédé jusqu'au bout, on n'aurait que des sorties OUI ou NON, la taille de l'algorithme serait souvent considérable et même supérieure du point de vue représentation mémoire à celle de la mémoire initiale.

Si on nomme M_A la mémoire occupée par le prétraitement, M_d la mémoire initiale prévue sans prétraitement. A chaque étape de l'algorithme de construction du prétraitement, M_A augmente d'une quantité que l'on peut considérer être le coût de l'opérateur supplémentaire et M_A diminue

du gain supplémentaire réalisé par cet opérateur. En fait, un meilleur algorithme de construction du prétraitement serait de choisir à chaque étape l'opérateur tel que le total (gain - coût) soit minimal.

Si on définit : M_T la mémoire totale nécessaire à chaque étape

$$M_T = M_A + M_O - \text{gain général}$$



Il suffit de choisir comme prétraitement l'opérateur obtenu au moment où la mémoire totale nécessaire recommence à croître, c'est-à-dire au moment où il n'est plus possible de trouver un opérateur tel que pour cet opérateur le coût soit supérieur au gain.

B. *ALGORITHMES DE RECHERCHE*

La notion de prétraitement permet d'extraire, autant que possible avec des opérateurs connus le "maximum de structure" de l'ensemble de données X. Nous disposons ensuite de plusieurs ensembles X_i que nous pouvons supposer chacun uniforme (sans structure particulière).

Nous cherchons dans cette partie B à décrire les algorithmes de recherche existants qui n'utilisent aucune notion particulière sur la structure des données (sauf quelquefois qu'elles sont constituées de mots sur un alphabet mais on peut toujours se ramener à ce cas).

1. *Méthode séquentielle*

C'est la plus simple, pratiquement la seule disponible lorsqu'on se sert de bandes magnétiques (très courant en raison du prix très faible de cette mémoire).

On range les valeurs les unes après les autres dans la mémoire et on les recherche en balayant la table ainsi constituée. Les divers coefficients peuvent ainsi être définis en moyenne.

$$\alpha = 1 \begin{cases} \beta = [X]/2 \\ \gamma = [X] \\ \gamma' = 1 \\ \delta = [X]/2 \end{cases}$$

On remarque que le coefficient γ' est égal à 1 s'il n'y a aucune vérification de double : il suffit d'ajouter une valeur au bout de la table. Cette méthode est très lourde pour les gros fichiers, en effet si

par exemple, on dispose d'un fichier de 1 000 valeurs il faudra 500 accès en moyenne pour en retrouver une. Il faudra 1 000 accès pour montrer qu'une valeur n'est pas dans la mémoire car on ne le saura que lorsqu'on aura parcouru toute la mémoire.

Il est possible d'améliorer le coefficient γ en supposant le fichier ordonné ce qui donne

$$\alpha = 1 \begin{cases} \beta = [X]/2 \\ \gamma = [X]/2 \\ \gamma' = [X]/2 \\ \delta = [X]/2 \end{cases}$$

On sait en effet qu'une valeur n'est pas dans la mémoire lorsqu'on a trouvé une valeur supérieure à la valeur cherchée. Ceci se fait bien sûr au détriment du coefficient γ' car, pour ajouter une nouvelle valeur en conservant l'ordre, il faut déplacer en moyenne la moitié du fichier.

Cette méthode est beaucoup plus efficace lorsqu'elle n'opère pas sur une seule valeur x mais sur tout un groupe. Il faut en effet un seul balayage pour retrouver toutes les valeurs cherchées et le nombre moyen d'accès par valeur s'en trouve réduit automatiquement.

C'est pour cette raison qu'en gestion, des traitements tels les mises à jour ou les fusions se font sur des fichiers gérés très souvent en séquentiel.

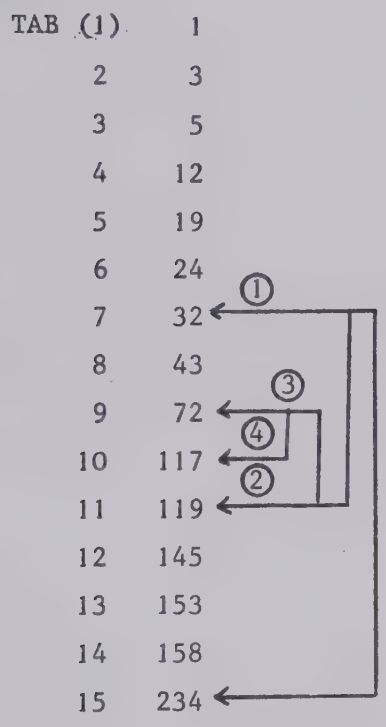
2. *Méthode dichotomique*

C'est aussi une méthode très connue. On suppose ici le fichier ordonné par exemple en ordre croissant et rangé tel quèl dans une table.

On compare ensuite la valeur à rechercher avec celle du milieu de la table, ce qui indique dans quelle partie, inférieure ou supérieure, se trouve la valeur en question puis on recommence avec la sous table ainsi sélectionnée.

Exemple :

Soit la table suivante :



Soit à chercher la valeur 117.

1. On commence par comparer 117 à la valeur du milieu de TAB soit TAB (7) = 32. La valeur 117 est supérieure à 32, si elle existe dans la table elle ne peut donc se trouver que dans la partie TAB (8) → TAB (15).

2. On compare 117 à la valeur du milieu de cette sous table soit TAB $(\frac{8 + 15}{2})$ = TAB (11) = 119. La valeur 117 est inférieure à 119 elle ne peut donc se trouver que dans la partie TAB (8) → TAB (10)

3. On compare 117 à la valeur du milieu de cette sous table soit TAB $(\frac{8 + 10}{2})$ = TAB (9) = 72. La valeur 117 est supérieure à 72 et donc ne peut se trouver que dans la partie TAB (10) → TAB (10).

4. On compare TAB (10) = 117 et 117 et on trouve la valeur cherchée.

Remarquons que si on avait cherché la valeur 118 par exemple, on aurait suivi le même cheminement, on aurait comparé la valeur TAB (10)= 117 avec 118, on aurait donc conclu que la valeur doit se trouver dans la sous table TAB (11) → TAB (10) de longueur - 1 ce qui est impossible. De la même façon on se serait donc aperçu que 118 n'existe pas dans la mémoire.

A chaque étape, la taille de la sous mémoire est divisée approximativement par 2. On en conclut qu'il faudra en moyenne $\text{Log}_2 [X]$ accès pour retrouver une valeur (en fait un peu moins car il est possible de trouver une valeur qui soit une charnière entre deux sous tables).

On peut donc en déduire les coefficients suivants :

$$\alpha = 1 \begin{cases} \beta = \text{Log}_2 [X] & \text{plutôt moins} \\ \gamma = \text{Log}_2 [X] & \text{plutôt plus} \\ \gamma' = \text{Log}_2 [X] + [X]/2 & \\ \delta = \text{Log}_2 [X] + [X]/2 & \end{cases}$$

Le terme à ajouter [X]/2 dans les deux derniers coefficients est normal car pour ajouter ou enlever une valeur, il faut tout d'abord la trouver (donc Log_2 [X] accès) ou trouver la place qu'elle devrait occuper puis déplacer ensuite en moyenne la moitié du fichier [[X]/2 accès].

C'est une méthode intéressante lorsque *l'ensemble des valeurs X est figé*, c'est-à-dire qu'il n'évolue pas, qu'il n'y a ni insertion ni suppression de nouvelles valeurs. Par exemple pour un fichier de 100 000 valeurs il faudra à peu près 17 accès pour retrouver une valeur alors que dans la méthode séquentielle, il en fallait environ 50 000. C'est aussi une méthode qui est très bonne lorsque *l'ensemble X tient entièrement en mémoire centrale*. Par contre, si le fichier est sur une mémoire externe, comme on passe d'un bout à l'autre du fichier à chaque test, chaque comparaison correspondra à une entrée-sortie réelle. Le facteur de groupage des informations sur la mémoire externe n'intervient pas, ce qui est dommage. Ce danger est particulièrement sensible lorsqu'on utilise une mémoire virtuelle. Lorsque le fichier est très gros, il tient certainement sur un grand nombre de pages et les précédentes raisons émises font qu'il y a un incessant transfert de pages pendant la recherche sans que le programmeur s'en rende compte donc au détriment des performances.

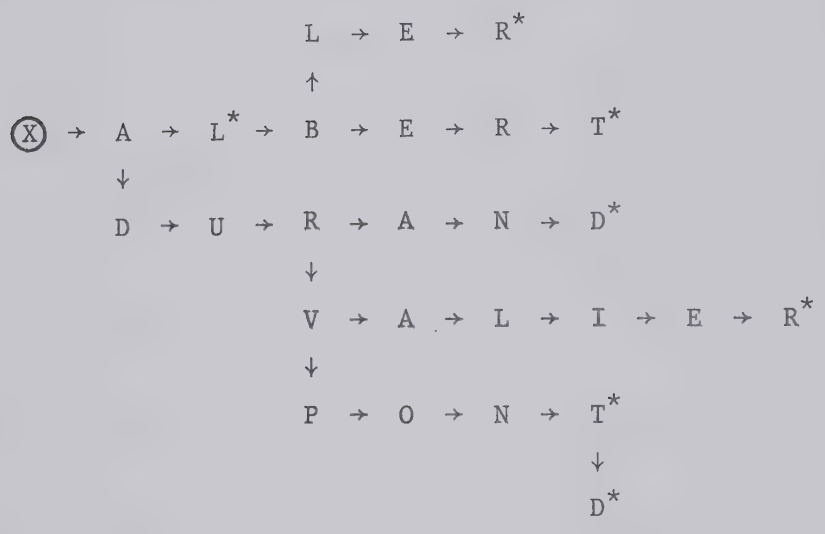
3. *Méthode des listes*

Cette méthode utilise le fait que les valeurs sont constituées de mots bâtis sur un alphabet. Elle peut s'appliquer à des données qui ne sont pas de longueur fixe. Elle consiste à représenter l'ensemble de mots X par un graphe dont les sommets sont les lettres des mots de X.

Par exemple, supposons l'alphabet usuel et l'ensemble de mots suivant :

ALBERT
DURAND
DUVAL
AL
DUPONT
DUVALIER
ALLER
DUPOND

Le graphe permettant de représenter cet ensemble peut avoir la forme suivante



Lorsque deux mots ont le même ensemble de premières lettres communes, on ne les écrit qu'une seule fois.

Les flèches horizontales sont nommées *successeurs* car chaque lettre reliée à une autre lettre par une telle flèche est le successeur de cette autre lettre dans au moins un des mots de X.

Les flèches verticales sont nommées *alternants* car elles indiquent que la lettre ayant pour successeur direct le premier alternant peut avoir n'importe lequel des alternants comme successeur dans l'ensemble de mots X

Il est nécessaire d'avoir un marqueur permettant d'indiquer la fin d'un mot (* dans notre exemple) car sinon les mots existants dans X, entièrement contenus dans d'autres mots de X (par exemple DUVAL et DUVALIE n'apparaîtraient pas.

Pour représenter une telle structure en mémoire, on affecte une cellule composée d'un ou plusieurs mots de la mémoire suivant les machines à chaque sommet du graphe.

Chaque cellule peut être représentée ainsi

(ad, car, suc, alt, fc)

ad : est l'adresse de la cellule, elle n'est pas physiquement contenue dans la cellule mais sert à la repérer.

car : est le caractère représentant le noeud du graphe.

suc : est l'adresse de la cellule successeur, cette partie représente la flèche horizontale de notre dessin. Lorsqu'il n'y a pas de successeur on admet une valeur privilégiée (par exemple 0) qui indique ce fait.

alt : est l'adresse de la cellule alternante lorsqu'elle existe sinon ce pointeur contient une valeur privilégiée (par exemple 0).

fc : est une marque indiquant la fin de mot. Il suffit d'un bit ayant la position 1 si fin de chaîne, 0 sinon.

Notre exemple se représenterait donc en mémoire de la façon suivante :

(1, A, 2, 7, 0)	(10, 1, 11, 0, 0)	(19, T, 0, 26, 1)
(2, L, 3, 0, 1)	(11, N, 12, 0, 0)	(20, I, 21,0, 0)
(3, B, 4, 23, 0)	(12, D, 0, 0, 1)	(21, E, 22, 0, 0)
(4, E, 5, 0, 0)	(13, V, 14, 16, 0)	(22, R, 0, 0, 1)
(5, R, 6, 0, 0)	(14, A, 15, 0, 0)	(23, L, 24, 0, 0)
(6, T, 0, 0, 1)	(15, L, 20, 0, 1)	(24, E, 25, 0, 0)
(7, D, 8, 0, 0)	(16, P, 17, 0, 0)	(25, R, 0, 0, 1)
(8, U, 9, 0, 0)	(17, 0, 18, 0, 0)	(26, D, 0, 0, 1)
(9, R, 10, 13, 0)	(18, N, 19, 0, 0)	

La méthode de recherche est simple, à partir de la cellule initiale, tant que les caractères sont identiques à ceux du mot cherché, on remonte la liste des successeurs, si on trouve un caractère différent, on cherche parmi les alternants de ce caractère si on en trouve un identique à celui cherché ; si on en trouve un, on continue de même dans les successeurs de cette nouvelle cellule sinon on déclare que le mot n'est pas dans la mémoire. Lorsque dans ce cheminement on trouve une cellule dont le caractère est identique au dernier caractère du mot à chercher on vérifie si cette cellule est une fin de chaîne et on dit qu'on a trouvé le mot cherché.

Par exemple pour chercher DUVAL

1. La première cellule est (1, A, 2, 7, 0) la lettre D ≠ A DUVAL ↑

2. On cherche un alternant de A (7, 0, 8, 0, 0) égalité de caractères

3. On passe au successeur de D (8, U, 9, 0, 0) égalité DUVAL ↑

4. Puis à une autre successeur (9, R, 10, 13, 0) lettre V ≠ R DUVAL ↑

5. On cherche un alternant de R (13, V, 14, 16, 0) égalité

6. Puis un successeur de V (14, A, 15, 0, 0) égalité DUVAL ↑

7. Enfin un dernier successeur (15, L, 20, 0, 1) égalité DUVAL ↑

C'est bien une fin de chaîne donc DUVAL est dans la mémoire.

Dans une telle organisation, il est souvent plus rapide de montrer qu'une valeur n'est pas dans la mémoire car on arrivera très rapidement à un alternant inexistant par exemple DALI

1. Première cellule	(1, A, 2, 7, 0)	lettre D ≠ A	DALI ↑
2. Alternant	(7, 0, 8, 0, 0)	égalité	
3. Successeur	(8, U, 9, 0, 0)	lettre U ≠ A	DALI ↑

Pas d'alternant, donc DALI n'est pas dans la mémoire.

Il est très facile d'ajouter un nom dans la mémoire. Il suffit de modifier une seule cellule et d'en ajouter un nombre réduit.

Par exemple, DUPRAT. On modifie (17, 0, 18, 0, 0) en (17, 0, 18, 27, 0) et on ajoute les cellules suivantes :

(27, R, 28, 0, 0)
(28, A, 29, 0, 0)
(29, T, 0, 0, 1)

Il est aussi simple d'enlever un nom de la mémoire. Par exemple ALLER. On modifie (3, B, 4, 23, 0) en (3, B, 4, 0, 0) et c'est tout, c'est-à-dire qu'on supprime virtuellement la chaîne L → E → R c'est-à-dire les cellules 23, 24, 25.

En fait, ces cellules ne sont pas perdues car il est possible de les chaîner avec une réserve de cellules libres et de les réutiliser lorsqu'arrivent de nouveaux mots.

On remarque donc la grande souplesse de cette méthode puisque les quatre coefficients ensembles (α, γ, γ', δ) semblent assez performants Pour des fichiers qui évoluent rapidement (avec des insertions et des suppressions) elle est très recommendable. Essayons d'évaluer ces divers coefficients sur un ensemble important de 20 000 noms de 6 lettres en moyenne.

Nous admettrons que nous avons 26 alternants au niveau de la 1ère lettre, 15 alternants en moyenne au niveau de la seconde, 10 à la 3ème, 5 à la 4ème, 1 à la 5ème et 0 à la 6ème lettre.

- Pour β , nous aurons à explorer environ 13 alternants en moyenne au niveau de la 1ère lettre (tout se passe comme si on avait des recherches séquentielles au niveau des alternants) 7,5 alternants au niveau de la 2ème puis 5, 3,5 et deux successeurs.

Soit $\beta = 13 + 7 + 5 + 3 + 2 = 30$ accès

- Pour γ, nous aurons

13 + 15 = 28 accès si la réponse est non au deuxième niveau

13 + 7 + 10 = 30 accès si la réponse est non au troisième niveau

13 + 7 + 5 + 5 = 30 accès au quatrième niveau

Soit approximativement $\gamma \simeq 30$.

- γ' et δ donnent sensiblement le même nombre 30.

Donc $\beta \simeq \gamma \simeq \gamma' \simeq \delta = 30$

Ces coefficients, pour β et γ, sont relativement élevés. En effet, on obtient les mêmes performances avec une recherche dichotomique portant sur un fichier d'un milliard d'éléments ($2^{30} \simeq 10^9$) ! et sur 20 000 valeurs la méthode dichotomique donne un nombre d'accès de l'ordre de 15. Il est heureusement possible d'améliorer la méthode des listes par exemple en ne créant pas une seule cellule d'entrée (1, A, x, y, 0) mais une table d'entrées de 26 cellules ordonnées suivant le caractère (que l'on peut omettre alors). On peut alors entrer dans cette table par indexation sur la représentation binaire du premier caractère. Le nombre d'alternants à parcourir en moyenne au niveau de la première lettre se ramène de 13 à 1 et les divers coefficients

$$\beta = \gamma = \gamma' = \delta \simeq 17$$

Ce qui compte du point de vue nombre d'accès dans cette méthode, c'est le nombre d'alternants à parcourir à chaque niveau. Nous venons de voir un moyen efficace de résoudre ce problème au premier niveau. On peut imaginer divers moyens aux niveaux suivants (dichotomies ou Has Coding sur de petites mémoires...) qui n'amélioreront que légèrement les résultats en compliquant singulièrement l'algorithme de recherche et surtout au détriment du coefficient α qui n'est déjà pas très élevé.

En effet, dans l'exemple du fichier de 20 000 noms, il faudrait environ 20 000 x 6 caractères x 5 bits = 600 000 bits dans la méthode séquentielle (ou dichotomique) alors que dans la méthode des listes simplifiée, il faut : environ 60 000 cellules donc 26 bits par pointeur Chaque cellule est composée de 2 pointeurs, 1 caractère, un bit de fin de chaîne soit 16 x 2 + 5 + 1 = 38 bits. 38 x 60 000 bits seront donc nécessaires et donc :

$$\alpha = \frac{600\ 000}{38 \times 60\ 000} \simeq 0{,}25$$

C'est-à-dire qu'il faut environ *4 fois plus* de mémoire dans la méthode par liste que dans la méthode séquentielle.

On admet généralement que pour des méthodes de listes plus évoluées il faut entre 3 et 7 fois plus de mémoire que dans la méthode séquentielle.

Il faut donc faire attention lorsqu'on choisit une mémoire organisée en structure de listes. Si le fichier est figé, la méthode dichotomique est en général meilleure jusqu'à un nombre de valeurs d'environ 10 000, elle est surtout plus simple et beaucoup plus économique du point de vue mémoire. La méthode de listes présente le même inconvénient que la méthode dichotomique en ce qui concerne les mémoires externes. Passer d'une cellule à son successeur ou son alternant peut provoquer à chaque fois une entrée sortie ou un changement de page.

Pour des fichiers variant faiblement ou étant uniquement en augmentation, nous verrons qu'il existe des méthodes bien plus performantes que les listes. Par contre, ces méthodes (Hash Coding, méthode approximative) ne sont pas très souples en ce qui concerne la suppression fréquente de valeurs. Là, semble-t-il, est le principal intérêt de la méthode de listes dans le problème du rangement en mémoire.

4. *Méthode directe*

C'est une méthode que nous avons utilisée dans un de nos premiers exemples. On a un ensemble X de valeurs prises parmi N. On réserve un tableau TAB de [N] éléments de 1 bit chacun. Si $x \in X$ on met un 1 à l'adresse TAB(x), si $x \in X$ on inscrit un 0.

Exemple : Soit l'ensemble X suivant

1, 4, 12, 7, 5, 19, 3, 8

valeurs prises parmi l'ensemble N = [1 20].

Cette organisation donne le tableau suivant :

TAB	1	1
	0	2
	1	3
	1	4
	1	5
	0	6
	1	7
	1	8
	0	9
	0	10
	0	11
	1	12
	0	13
	0	14
	0	15
	0	16
	0	17
	0	18
	1	19
	0	20

Soit 20 bits alors qu'il en faudrait une trentaine si on rangeait les nombres 1'un après 1'autre avec codage approprié.

C'est la méthode idéale. Il faut *un seul accès* pour retrouver une valeur, un seul accès pour montrer qu'une valeur n'est pas dans la table, un seul accès pour ajouter ou supprimer une valeur

$$\beta = \gamma = \gamma' = \delta = 1$$

De plus la méthode est on ne peut plus simple.

Mais

1. Lorsque [X] est très petit devant [N] , ce qui arrive très couramment dans la pratique, une quantité de [N] bits peut être fantastique et le coefficient α ridiculement petit.

Par exemple, pour notre fichier de noms de 6 caractères. L'ensemble N est composé de $(27)^6$ éléments soit $\simeq 10^{12}$. Pour notre fichier de 20 000 mots on aurait :

$$\alpha = \frac{20\,000 \times 6 \times 5}{10^{12}} \simeq 10^{-6}$$

et il est impensable dans ce cas d'utiliserune telle méthode. Ceci prouve qu'une méthode peut être idéale en théorie et impraticable en informatique dans certains cas.

2. Très souvent, on ne range pas des valeurs uniquement pour pouvoir répondre oui ou non à la question : x appartient-il à X ? Chaque valeur possède un indicatif (l'adresse d'un enregistrement sur un disque par exemple) que l'on désire obtenir si la réponse est oui. Il faut donc, en regard de chaque valeur, conserver cet indicatif. Dans cette méthode qui contient beaucoup de "trous" (bits nuls) on perdra beaucoup de place supplémentaire car on ne pourra rien inscrire à la place de l'indicatif qui sera de toute façon réservée. (On pourrait bien sûr admettre que la place de l'indicatif puisse être récupérée pour un autre fichier géré en liste si les cellules peuvent tenir mais on mélange ici plusieurs problèmes).

Il ne faut tout de même pas oublier cette méthode, elle peut être utile plus souvent qu'on ne le pense lorsqu'on a un petit nombre de valeurs possibles comme dans l'exemple du début. En fait, elle devient intéressante dès que [X] Log [N] est voisin de [N].

Dès que $[N] < 2\,[X]\ \text{Log}\,[N]$

ou encore $[N] > \dfrac{[N]}{2\ \text{Log}\ N}$

Exemples

[N] = 100	[X] > 7
[N] = 1 000	[X] > 50
[N] = 1 000 000	[X] > 25 000

5. *"Hash Coding"*

a. Présentation, généralités

Cette méthode que l'on trouve aussi sous les noms divers de Hash Code, fonction de hachage, transformation clef-adresse,a sous sa forme primitive une quinzaine d'années d' existence . Elle est très simple, très performante et pourtant relativement peu employée. Ceci vient du fait que si les autres méthodes paraissent utiliser une certaine idée de la structure des valeurs (l'ordre dans la méthode dichotomique ; la notion de mots dans la méthode des listes) cette méthode, elle, consiste à ne se servir en rien de la structure pouvant exécuter. En général, les utilisateurs désirant utiliser un fichier ont eux-mêmes une certaine idée de la structure de leur fichier et il est quelquefois difficile d'admettre que cette connaissance de la structure ne sert à rien pour le problème de rangement des valeurs.

L'idée essentielle du Hash Coding est de trouver une fonction h, qui permette de transformer la valeur x en une adresse d'une mémoire M. On range ensuite la valeur x à cette adresse.

Le nombre de valeurs possibles [N] étant bien plus grand que le nombre d'adresses de M il est possible que plusieurs valeurs x_1, x_2 donnent la même adresse m par la transformation h.

$$h(x_1) = h(x_2) = m$$

On dit dans ce cas qu'il y a collision en m ou encore que les valeurs x_1 x_2 ... sont synomymes.

Tout le problème du Hash Coding est :

1. de trouver une "bonne fonction h" qui donne le plus petit nombre possible de collisions, c'est-à-dire une fonction h qui disperse bien les valeurs de N. Ceci pourrait s'écrire
   $\forall m, \in M, \forall m_2 \in M\ [h^{-1}(m_1)] = [h^{-1}(m_2)]$

2. de résoudre le problème des collisions.

Exemple :

Donnons comme exemple, une mémoire M de 15 mots et un ensemble X de 10 nombres : 13, 51, 74, 99, 3, 1, 42, 7, 28, 44 pris au hasard.

Prenons comme fonction h(x) une fonction couramment employée dans les méthodes de Hash Coding.

$$h(x) = \text{Reste } (x \ / \ 15)$$

Pour résoudre les collisions, nous chercherons, lorsqu'il y a collision à placer la valeur 4 adresses plus loin, et ainsi de suite de 4 en 4, si le mot à cette adresse est déjà occupé. Pour définir l'occupatio

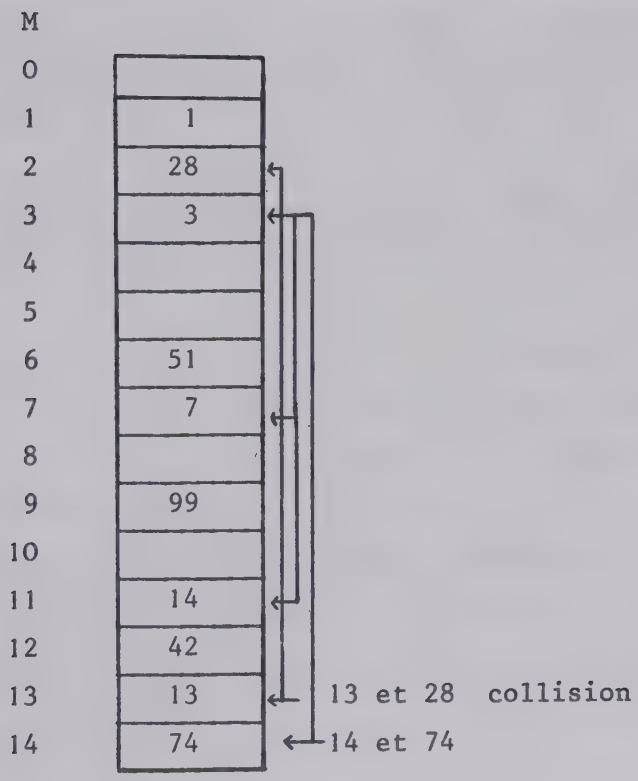
mémoire et la non occupation mémoire, nous supposerons que la valeur zéro n'existe pas dans N.

- h(13) = 13 on range 13 à l'adresse 13 inoccupée en 1 seul accès

- h(51) = 6 on range 51 à l'adresse 6 inoccupée en 1 seul accès

- h(74) = 14 74 se range à l'adresse 14 en 1 seul accès

- h(99) = 9 99 se range en 9 : 1 seul accès.



- h(3) = 3 — 3 se range en 3 : 1 accès
- h(1) = 1 — 1 se range en 1 : 1 accès
- h(42) = 12 — 42 se range en 12 : 1 accès
- h(7) = 7 — 7 se range en 7 : 1 accès
- h(28) = 13 — la mémoire 13 est occupée. On essaye en 13 + 4 = 17 = 2 modulo 15 qui est une mémoire inoccupée ou on range la valeur 28 : 2 accès
- h(44) = 14 — la mémoire 14 est occupée. On essaye en 13 + 4 = 18 = 3 modulo 15 qui est une mémoire occupée, on essaye en 7 + 4 = 11 qui est inoccupée et on range la valeur 14 : 4 accès.

Il a fallu 14 accès en tout pour ranger les 10 valeurs soit un coefficient moyen de 1,4. Comme on retrouve chaque valeur exactement de la même façon, on en déduit que pour cet exemple $\beta = 1,4$ ce qui est excellent. On s'aperçoit aussi sur cet exemple, qu'au début il est très facile de ranger les valeurs. On arrive très vite sur une mémoire inoccupée. Par contre, vers la $10^{ème}$ il faut déjà 4 accès pour en ranger une. On conçoit très bien que si on essayait de ranger une $11^{ème}$ valeur, puis une $12^{ème}$ etc... on aurait de plus en plus de mal à les ranger rapidement. Ceci est une des caractéristiques du "Hash Coding". Pour que la méthode soit bonne, il faut qu'il y ait plus de mémoires que de valeurs dans l'ensemble X. Ceci indique que les performances du Hash Coding dépendent essentiellement du coefficient de remplissage α. On admet généralement que le Hash Coding est très bon jusqu'à $\alpha = 0,5$, bon jusqu'à $\alpha = 0,75$ et qu'il doit pas être employé au-dessus.

On s'aperçoit aussi sur cet exemple que le coefficient α se détèriore beaucoup plus vite que β (suivant α). En effet, on peut dire qu'une valeur n'est pas dans la mémoire que lorsqu'on arrive à une mémoire vide (inoccupée).

Dans l'exemple précédent, il faut

1 accès	pour les valeurs	telles que	h(x) = 0
2 accès	"	"	h(x) = 1
3 accès	"	"	h(x) = 2
4 accès	"	"	h(x) = 3
1 accès	"	"	h(x) = 4

etc...

soit en moyenne sur l'ensemble des valeurs de N

$$\gamma = 3,8$$

On conçoit bien que moins il y aura de mémoires inoccupées, plus il sera difficile de trouver une "trou" et plus le coefficient γ sera grand.

Il n'est pas difficile d'ajouter une valeur, mais pour l'ajouter il faut trouver une place vide pour y inscrire la valeur, on a donc

$$\gamma' = \gamma$$

Il est par contre beaucoup plus difficile d'enlever une valeur. Supposons que nous désirerions enlever la valeur 7, on a h(7) = 7 et le contenu de la mémoire 7 est 7. Il n'y a pas de problème et on remplace cette valeur par 0 *mais* si on cherche ensuite à nouveau la valeur 14, cette valeur n'est pas dans la mémoire. On a :

h(14) = 14 : collision, on essaye à l'adresse 14 + 4 = 3 [15]. En 3, il y a collision, on essaye en 3 + 4 = 7 qui contient la valeur 0. On en déduit que 14 n'est pas dans la mémoire et c'est faux.

Ceci tient au fait que les valeurs dans la mémoire ont un double rôle. Elles sont utilisées pour déterminer si cette valeur elle-même, est ou n'est pas dans la mémoire mais elles servent aussi par leur présence en tant que chaînage implicite.

Il faudrait donc lorsqu'on enlève une valeur, enlever la représentation de la valeur mais ne pas détruire la notion de présence. Ceci peut se faire en supposant qu'en plus du zéro une autre valeur (-1 par exemple) n'appartient pas à N et en remplaçant une valeur par -1 lorsqu'on l'enlève.

b. Etude générale

Etant donnée une suite de fonctions $(h_0, h_1, \ldots, h_i, \ldots)$ de X sur une mémoire adressable M.

L'algorithme dit de "Hash Coding" est le suivant :

1. i = 0
2. Calculer $h_i(x)$
3. Si cette adresse est inoccupée, ranger x, fin
4. Sinon i = i + 1 recommencer en 2.

On essaye donc successivement les adresses $h_1(x)$, $h_2(x)$, $h_3(x)$, ... jusqu'à ce qu'on puisse trouver une place où ranger la valeur. Pour la recherche, on fait de même jusqu'à identification.

Lorsque deux valeurs x et y aboutissent à la même adresse $h_i(x)$, on dit qu'il y a collision.

La suite de fonctions $(h_0, h_1, \ldots, h_i, \ldots)$ est en général donnée par récurrence : $h_i = \varphi(h_{i-1})$. On essaye d'avoir des fonctions qui dispersent aléatoirement les valeurs dans la mémoire, c'est pourquoi il est préférable qu'elles soient indépendantes.

- *Choix de la fonction* h_0

Un certain nombre de fonctions h_0 ont été introduites depuis une dizaine d'années mais très souvent, les personnes qui ont introduit de nouvelles fonctions n'ont que très peu essayé de les comparer entre elles. Deux études comparatives réelles existent seulement à notre connaissance, l'une par BUCHHOLZ (1963) et une autre beaucoup plus récente et très pertinente par V. Y. LUM, P. S. T. YUEN et M. DODD (1971) que nous allons utiliser ici.

LUM & co ont choisi un certain nombre de fichiers différents par leur taille (de 500 à 30 000 enregistrements) et par leur nature. Certaines sont numériques, d'autre alphabétiques sur des noms ou des mots, d'autres alphanumériques avec une structure complexe. Ils ont ensuite essayé les diverses fonctions h_0 définies ci-après avec la même méthode de résolution des collisions ; une recherche de une en une adresse à partir de l'adresse $h_0(x)$.

Fonctions h_0 étudiées :

1. Méthode de la division

$$h_0 = \text{reste (x / taille de M)}$$

C'est la méthode que nous avons utilisée dans notre exemple.

2. Sélection de caractères de la clef (Digit Analysis)

On suppose que la valeur (clef) est composée d'une suite de caractères Cette méthode suppose ensuite une étude statistique préalable sur les différents caractères qui composent les valeurs à ranger. On sélectionne par cette étude, les positions des caractères qui sont censés être significatifs des mots. On suppose pour chaque valeur, que la sélection des caractères prise à ces places choisies dans un ordre déterminé est suffisamment réduite pour donner une adresse "bien distribuée" dans l'ensemble des adresses. Par exemple, dans un ensemble de Noms, on choisira la première lettre, la troisième et la cinquième qui sont plus significatives que la seconde ou la quatrième dans l'ensemble des mots connus.

C'est une des seules méthodes essayant d'exploiter la connaissance de la distribution des clefs. Elle est à rapprocher d'une des méthodes dites "Approximatives" que nous verrons plus loin.

Bien qu'elle suppose une étude préalable des valeurs du fichier pouvant entraîner une perte de temps considérable et qu'elle ne soit pas (et de loin) une des meilleures, la fonction h_0 ainsi définie est très employée dans les entreprises et particulièrement en France.

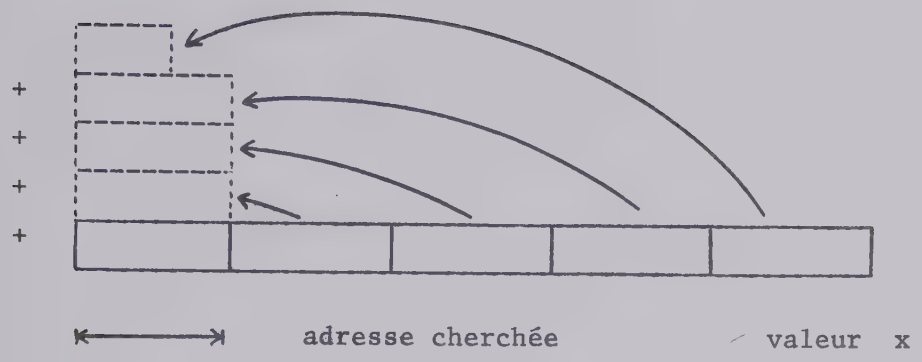
3. Moindre carré

La valeur x est multipliée par elle-même et la valeur de $h_0(x)$ est obtenue en choisissant une partie fixe du résultat de cette multiplication.

4. Compression

On découpe la représentation de la valeur x en petites parties de la taille de l'adresse cherchée et on additionne ces diverses parties.

Exemple



5. *Méthode de LIN (1963)*

On considère dans cette méthode que la valeur est exprimée suivant une base P et le résultat de $h_0(x)$ est pris modulo q^m. P et q sont premiers entre eux, m est un entier tel que q^m soit proche de la taille de la mémoire. On choisit aussi P et q tels que P = q + 1.

6. *Codage algébrique*

Tous les digits de la clef sont considérés comme étant les coefficients d'un polynome. Ce polynôme est alors divisé par un autre polynôme g(x) qui possède la propriété d'être invariant pour toutes les clefs d'un certain ensemble. Les coefficients du polynôme, reste de cette division, forment l'adresse cherchée. Cette méthode est basée sur la théorie des codes autocorrecteurs (PETERSON (1961)).

7. Théorique

Les résultats sont comparés avec ceux obtenus par PETERSON, W. Q. (1957) d'une façon théorique.

Supposons une mémoire de taille M qui contient déjà R valeurs. Si les valeurs sont aléatoirement placées dans la mémoire, le nombre moyen d'accès γ pour trouver une place vide est

$$\gamma = \frac{M + 1}{M + 1 - R}$$

Nous avons vu que pour ces méthodes, on a $\gamma = \gamma'$. Or

$$\beta = \frac{1}{[X]} \sum_{R=0}^{[X]-1} \gamma(R)$$

en fonction même du caractère séquentiel de construction de la mémoire, et donc

$$\beta = \frac{1}{[X]} \sum_{R=0}^{[X]-1} \frac{M + 1}{M + 1 - R}$$

ce qui peut s'écrire

$$\beta = \frac{M + 1}{[X]} \left(\sum_{k=1}^{M+1} \frac{1}{k} - \sum_{k=1}^{M-[X]+1} \frac{1}{k} \right)$$

Or, $\sum_{k=1}^{n} \frac{1}{k}$ peut être approximé par $\text{Log}_e\ n + 0{,}5772157$

et donc $$\beta = \frac{M + 1}{[X]} \left(\text{Log}_e\ (M + 1) - \text{Log}_e\ (M + 1 - [X])\right),$$

si $M + 1 - [X]$ n'est pas trop petit c'est-à-dire qi le coefficient α n'est pas trop proche de 1.

On suppose que la longueur de la représentation des éléments de N est identique à celle des mots de la mémoire. On peut écrire :

$$\alpha \simeq \frac{[X]}{M + 1}$$

et donc

$$\beta \simeq -\frac{1}{\alpha} \text{Log}_e (1 - \alpha)$$

$$\gamma \simeq \frac{1}{1 - \alpha}$$

Les résultats des comparaisons entre ces diverses fonctions g_0 sont donnés dans la Table 1. Seul, le coefficient β est étudié et on trouve, entre parenthèses, à coté de chaque moyenne, son écart quadratique moyen.

Une conclusion essentielle de cette étude est que *la meilleure* méthode est celle de *la division*, c'est-à-dire *la plus simple* et une des premières utilisée. La méthode par sélection, couramment employée est visiblement une des plus mauvaises.

Il est très réconfortant que la méthode la plus simple soit la meilleure (prendre le reste d'une division est une opération élémentaire sur une machine) mais il peut sembler inquiétant que pendant 12 ans, diverses méthodes aient été utilisées et publiées sans qu'il n'y ait jamais eu de comparaisons véritables sur ces fonctions. Nous ajouterons que dans le même temps que LUM & co nous avons aussi essayé diverses fonctions h_0 et étions arrivés à la même conclusion ce qui confirme ces résultats.

- TABLE I -

Etude de β

Méthodes	$\alpha = 0,5$	$\alpha = 0,6$	$\alpha = 0,7$	$\alpha = 0,8$
Division (1)	1,19 (0,09)	1,25 (0,10)	1,28 (0,10)	1,34 (0,07)
Sélection (2)	1,42 (0,26)	1,41 (0,27)	1,45 (0,29)	1,51 (0,26)
Moindre carré (3)	1,34 (0,03)	1,32 (0,02)	1,36 (0,03)	1,41 (0,03)
Compression (4)	1,38 (0,18)	1,52 (0,60)	1,40 (0,22)	1,48 (0,17)
Lin (5)	1,57 (0,59)	1,62 (0,65)	1,63 (0,61)	1,81 (0,74)
Codage algébrique (6)	1,35 (0,07)	1,30 (0,07)	1,36 (0,07)	1,40 (0,10)
Théorique (7)	1, 35	1,30	1,35	1,40

D'après V. Y. LUM, P. S. T. YUEN, M. DODD (1971)

Une autre constatation est sortie de ces études sur les fonctions h_0, c'est que dans le cas de la méthode de la division, *il n'est absolument pas nécessaire* de prendre le reste de la division de la valeur x par le plus grand *nombre premier* inférieur à la taille de la mémoire. Cette restriction introduite par BUCHHOLZ (1963) a fait le tour du monde et est même devenue une légende tenace.

BUCHHOLZ avait introduit cette restriction car il pensait qu'il pourrait se former des classes dans la mémoire et qu'il arrive qu'on ne puisse plus ranger de valeurs alors qu'il y a encore de la place dans la mémoire. Cette restriction tient si on essaye de remplir *complètement* la mémoire. En effet, lorsqu'il ne reste plus qu'une place vide, n'importe qu'elle valeur de N - X doit pouvoir arriver à cette place. Nous avons vu que les performances du Hash Coding se détèriorent sérieusement lorsque α est voisin de 1 au point d'être équivalent au séquentiel pour les dernières valeurs. Lorsqu'on choisit de gérer une mémoire par Hash Coding on ne doit pas obtenir un α supérieur à 0,8 (sinon ce n'est pas la peine de choisir cette méthode) et dans ce cas, si la mémoire est grande, il est inutile de prendre un nombre premier.

Nous avons d'ailleurs parfaitement confirmé ce fait. En effet, dans une des méthodes que nous développerons plus loin, nous avons généré automatiquement des "Hash Coding" successifs. La taille des diverses mémoires de rangement est déterminée aléatoirement et jamais nous n'avons eu de problème avec les performances. LUM & co ont d'ailleurs vérifié que des nombres non premiers donnaient très souvent des résultats très légèrement meilleurs que des nombres premiers, ce qui prouve qu'il n'y a aucune corrélation entre le fait que le diviseur soit premier et l'efficacité de la méthode (un nombre sur 10 est premier environ ce qui fait que 5 nombres sur 10 donnent en moyenne de meilleurs résultats).

- *Résolution des collisions*

Nous avons vu que la résolution des collisions consiste à choisir une fonction de récurrence φ sur l'ensemble des fonctions de "Hash Coding" $(h_0, h_1, \ldots, h_i, \ldots)$

$$h_i = \varphi(h_{i-1})$$

Nous avons déjà rencontré dans nos exemples une telle fonction

$$h_i(x) = h_{i-1}(x) + a \quad [M]$$

modulo la taille de la mémoire

ou encore

$$h_i(x) = h_0(x) + a * i \quad [M] \qquad (1)$$

Le terme a était égal à 1 dans l'étude comparative sur les fonctions h_0. Ce terme a est choisi arbitrairement, sa valeur n'a pas une grande importance mais si les valeurs restent groupées dans la mémoire, un "a" faible ne fera pas éclater ce groupe et ce peut être un désavantage. Par contre, un a trop élevé donnera une succession de tests à des endroits fort éloignés les uns des autres dans la mémoire et peut avoir une certaine influence lorsque le facteur de groupage est considéré, comme nous le verrons plus loin.

L'inconvénient de cette fonction de récurrence (1) est qu'il se produit souvent des classes de valeurs espacées de a en a dans la mémoire. Lorsqu'une recherche commence dans une de ces classes, il faut un grand nombre d'accès pour en sortir. Une deuxième formule de récurrence utilisée a été ensuite :

$$h_i(x) = h_0(x) + a * i + b * i^2 \quad [M] \qquad (2)$$

Le terme i^2 permet d'éliminer les classes génantes vues précédemment mais n'empêche pas que deux valeurs x_1 et x_2 telles que $h_0(x_1) = h_0(x_2)$ suivent le même cheminement de tests. La possibilité d'avoir d'autres sortes de classes subsiste. Une dernière amélioration de la formule (2) [BELL (1970)] permet d'éliminer aussi cette sorte de classe.

$$\boxed{h_i(x) = h_0(x) + a * i + b(x) * i^2 \ [M]} \qquad (3)$$

C'est cette dernière formule qui a donné les meilleurs résultats jusqu'à présent, le terme $b(x)$ permettant d'éliminer les classes provenant de valeurs possédant le même résultat par $h_0(x)$ dès le deuxième pas.

D'autres méthodes de résolution des collisions ont aussi été essayées. Certaines permettent aussi de remplir totalement la mémoire, ce qui n'est pas certain ici. Ces précautions sont toujours illusoires si α ne dépasse pas 0,8 et aucune ne semble avoir la simplicité et les performances de cette dernière. Comme fonction $b(x)$, il est possible d'utiliser le quotient x/M ce qui est très pratique si h_0 est la méthode de la division.

Nous avons expérimenté ces trois résolutions des collisions notées HC1, HC2, HC3 sur les fichiers suivants que nous utiliserons aussi dans les autres méthodes.

Fichier A1 : fichier de 5o nombres au maximum de valeurs inférieures à 1 000, choisis au hasard.

Fichier A2 : fichier de mots français codés dans une base 27 et cadrés à gauche (seuls les 8 premiers caractères sont envisagés).
Le fichier est d'environ 500 mots provenant d'un extrait de "Du côté de chez SWAM" de Marcel PROUST. Le fichier possède des répétitions (les mots courants) et le coefficient β de recherche est un coefficient modifié par une certaine utilisation de la mémoire. Les mots communs se répètent et ont été rangés parmi les premiers. Ils se retrouvent donc rapidement c'est pourquoi le coefficient β est si excellent dans ce fichier.

Fichier A3 : fichier de 2 000 nombres aléatoires donnés par la fonction standard de l'ordinateur (CDC 3600).

Les trois "Hash Coding" sont donnés par :

HC1 $\quad h_i(x) = \text{reste } (x/M) + 7 * i \quad$ Modulo M

HC2 $\quad h_i(x) = \text{reste } (x/M) + 7 * i + i^2 \quad$ Modulo M

HC3 $\quad h_i(x) = \text{reste } (x/M) + 7 * i + (x/M) * i^2 \quad$ Modulo M

Le tableau II donne les résultats. On constate bien que les diverses améliorations apportées à la résolution des collisions sont efficaces.

- TABLE II -

Fichier A1

α/β	HC1	HC2
0,2	1,30	1,50
0,3	1,33	1,53
0,4	1,65	1,80
0,5	1,76	2,08
0,6	1,97	2,40
0,7	2,00	2,46
0,8	2,57	2,75

Fichier A2

	HC1		HC2		HC3	
α	β	γ	β	γ	β	γ
0,2	1,00	1,06	1,00	1,05	1,00	1,05
0,3	1,02	1,23	1,02	1,25	1,02	1,23
0,4	1,05	1,32	1,10	1,38	1,05	1,37
0,5	1,14	1,40	1,17	1,42	1,11	1,41
0,6	1,15	1,70	1,16	1,73	1,11	1,53

Fichier A3

α/β	HC1	HC2	HC3
0,2	1,26	1,14	1,13
0,3	1,23	1,20	1,19
0,4	1,38	1,33	1,27
0,5	1,59	1,46	1,38
0,6	1,86	1,62	1,50
0,7	2,48	1,84	1,68
0,8	3,91	2,18	1,99

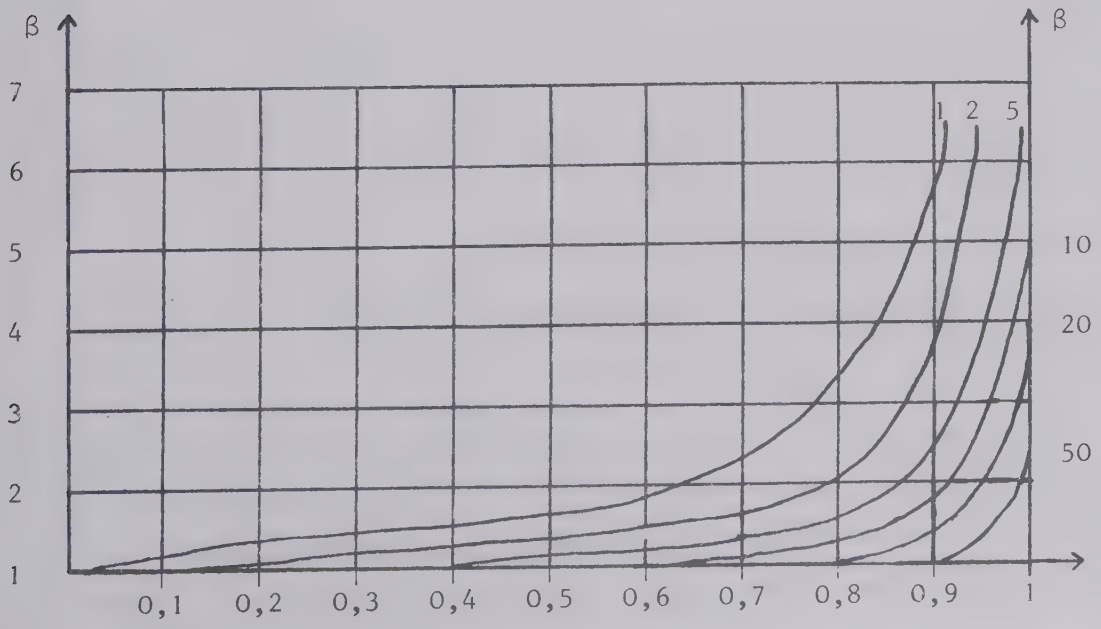
- *Influence du facteur de groupage*

Nous avons vu que lorsque nous utilisons des mémoires externes telles les disques, tambours, feuillets magnétiques... ce qui est souvent le cas, nous n'avons pas accès à une information unique mais à un groupe d'informations.

Il serait donc intéressant, lorsqu'on passe d'une valeur $h_i(x)$ à $h_{i+1}(x)$ que cette seconde adresse soit dans le même groupe obtenu par la première entrée/sortie. Le temps de recherche en mémoire centrale est supposé négligeable par rapport aux entrées/sorties réelles. Dans ce cas, le coefficient β réel se trouve considérablement amélioré pour être même très voisin de 1.

C'est pour celle raison, que la technique de résolution des collisions qui consiste à se déplacer faiblement (a petit) autour de la position initiale $h_0(x)$ est intéressante. Il est possible aussi de remplacer le terme "modulo la taille de la mémoire" par "modulo le groupe". On effectue le calcul réel donné par la formule de résolution mais on ne sort pas du groupe ; ou alors lorsqu'il est plein dans le groupe suivant.

La figure 3 montre l'influence de la taille d'un groupe sur le coefficient β en entrées/sorties réelles. Le fichier allant jusqu'à 10 000 nombres aléatoires.

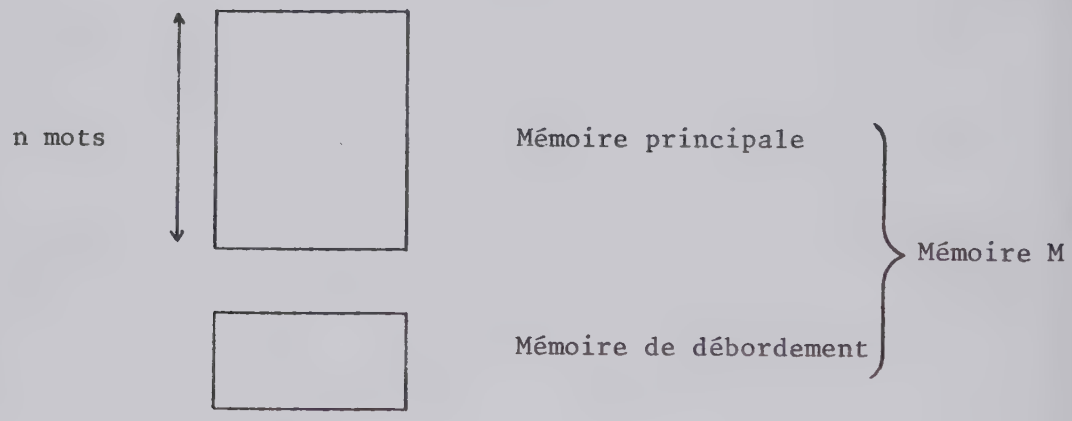


D'après W. W. PETERSON (1957)

- *Résolution des collisions à l'aide d'une mémoire de débordement*

Cette méthode est une combinaison de la méthode de "Hash Coding" et de la méthode des listes. La valeur x pour laquelle il y a collision est rangée dans une mémoire de débordement de la première.

Nous définirons n comme étant la taille de la première mémoire et X (au lieu de [X] pour simplifier la formulation) le nombre de valeurs à ranger.



Posons $k = X/n$.

Supposons que la répartition des adresses obtenues par $h(x)$ soit uniforme.

Si on range les X valeurs, la probabilité en une adresse donnée m d'avoir une collision d'ordre P (l'ordre d'une collision étant le nombre de valeurs de X entrant en collision en m) peut aisément être calculée. Notons $\theta(m)$ l'ordre d'une collision en m.

Parmi les X valeurs il y a C_X^P différents groupes susceptibles de former des collisions d'ordre P. Pour chacun de ces groupes, les $X - P$ valeurs restantes peuvent se ranger n'importe ou dans les $n - 1$ cases restantes.

Soit $C_X^P (n - 1)^{X-p}$ solutions sur n^X possibles.

Donc $P(\theta(m) = P) = \dfrac{C_P^X (n - 1)^{X-P}}{n^X}$

La table contenant n mots, l'espérance du nombre de cases occupées par une collision d'ordre P sera donc :

$$C(P) = \frac{n\, C_X^P}{(n - 1)^P} \left(\frac{n - 1}{n}\right)^X$$

Cette formule valable pour $P = 0$ donne pour les premières collisions :

$$C(0) = n \left(\frac{n - 1}{n}\right)^X$$

$$C(1) = X \left(\frac{n - 1}{n}\right)^{X-1}$$

$$C(2) = \frac{k}{2} (X - 1) \left(\frac{n - 1}{n}\right)^{X-2}$$

Ces trois collisions sont les plus intéressantes. Elles caractérisent, nous le verrons, la grande majorité des valeurs de X. Lorsque n devient très grand $(n \to \infty)$ ces trois valeurs sont équivalentes respectivement à

$$C(0) \sim n \times \frac{1}{e^k}$$

$$C(1) \sim k^n \times \frac{1}{e^k}$$

$$C(2) \sim \frac{k^2}{2} n \times \frac{1}{e^k}$$

Car $$(\frac{n-1}{n})^{n_1} = (\frac{n-1}{n})^{k_n} \to \frac{1}{e^k}$$

Ceci signifie que, lorsque n est grand, la proportion de mémoires inoccupées **ou de** mémoires ayant une collision d'ordre 1, 2 ou P ne dépend que de la valeur de k. k est le rapport entre le nombre d'informations à ranger et la taille de la mémoire principale (une sorte de coefficient qui ne tiendrait pas compte de la mémoire de débordement).

Quelques valeurs calculées sont données dans la table suivante :

	C(0)	C(1)	C(2)
k = 1/2	61 %	30,5 %	7,5 %
k = 1	37 %	37 %	18,5 %
k = 2	13,5 %	27 %	27 %

Les pourcentages sont des pourcentages de n

k = 1 donne le maximum pour C(1).

Cette table signifie que si on prend une mémoire principale n deux fois plus grande (k = 1/2) que la taille du Fichier X, (30,5 + 2 x 7,5) x 2 = 91 % des valeurs de X font parties de collisions d'ordre inférieur ou égal à 2. Les performances ne peuvent être qu'excellentes.

Si on prend une mémoire n de la taille du fichier (k = 1) 37 + 2 x 18,5 = 74 % des valeurs font parties de collisions d'ordre inférieur ou égal à 2.

Si on prend une mémoire n de la moitié de la taille du fichier (k = 2) 1/2(27 + 2 x 27) = 40,5 % des valeurs de X font parties de collisions d'ordre inférieur ou égal à 2 et les performances ne doivent pas être très bonnes.

La proportion $\frac{n - C(0)}{X}$ est aussi intéressante car elle définit la proportion de valeurs qui seront atteintes en un seul accès.

k = 1/2	78 % de X
k = 1	63 % de X
k = 2	43 % de X

- *Influence d'une mémoire de débordement sans pointeurs*

Ignorons dans une première phase l'existence des pointeurs. Supposons que nous n'utilisons que la mémoire de débordement nécessaire pour ranger les valeurs entrant en collision avec des valeurs déjà rangées.

M = n + nombre de valeurs restant à ranger

Soit $M = n + [X - (n - C(o))]$

Ou encore $\boxed{M = n(k + (\frac{n - 1}{n})^{kn})}$

Soit quand n est grand $M \sim n(k + e^{-k})$. Comme nous avons défini $\alpha = \frac{[X]}{M}$

$$\alpha = \frac{k}{k + (\frac{n-1}{n})^{k_n}} \qquad \boxed{\alpha \sim \frac{k}{k + e^{-k}}}$$

Exemple

$k = 1/2$	$M = 1,11$ $n = 2,22$ X	$\alpha = 45$ %
$k = 1$	$M = 1,37$ $n = 1,37$ X	$\alpha = 73$ %
$k = 2$	$M = 2,13$ $n = 1,07$ X	$\alpha = 94$ %

Le nombre moyen d'accès β est fonction du nombre de collisions et de leur ordre P.

$$\beta = \frac{1}{X} \sum_{P=1}^{X} \frac{(P+1)^P}{2} C(P)$$

car nous avons P valeurs par collisions d'ordre P et $\frac{P+1}{2}$ recherches en moyenne parmi ces collisions.

Soit $$\beta = \frac{1}{2X} \sum_{P=1}^{n_1} \frac{P(P+1)\, C_X^P\, (n-1)^{X-P}}{n^{X-1}}$$

ou encore $$\beta = \frac{1}{2} \frac{(n-1)^{X-1}}{n} \sum_{e=0}^{X-1} (e+2)\, C_{X-1}^e\, (\frac{1}{n-1})^e$$

Posons $$I = \sum_{e=0}^{X-1} (e+2)\, C_{X-1}^{e} \left(\frac{1}{n-1}\right)^{e}$$

$$= 2 \sum_{e=0}^{X-1} C_{X-1}^{e} \left(\frac{1}{n-1}\right)^{e} + \sum_{e=1}^{X-1} e\, C_{X-1}^{e} \left(\frac{1}{n-1}\right)^{e}$$

$$= 2 \left(\frac{n}{n-1}\right)^{X-1} + \left(\frac{X-1}{n-1}\right)\left(\frac{1}{n-1} + 1\right)^{X-2}$$

$$= \frac{(n)^{X-2}}{(n-1)^{X-1}} (2n + X - 1)$$

Donc $$\beta = \frac{1}{2} \left(\frac{n-1}{n}\right)^{X-1} \frac{n^{X-2}}{(n-1)^{X-2}} (2n + X - 1)$$

et finalement $$\beta = 1/2 \, \frac{X(k+2) - k}{X}$$

Donc pour un grand fichier (X grand)

$$\beta \sim 1 + \frac{k}{2}$$

Exemples

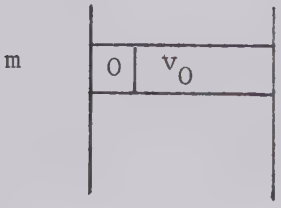
$k = \frac{1}{2}$	$\alpha = 45\ \%$	$\beta = 1,25$
$k = 1$	$\alpha = 73\ \%$	$\beta = 1,50$
$k = 2$	$\alpha = 94\ \%$	$\beta = 2$

En comparaison, pour la meilleure méthode de Hash Coding sans débordement, les coefficients β pour les mêmes valeurs de α étaient 1,32 pour α = 45 %, 1,78 pour α = 73 %. Ces résultats semblent bien meilleurs et on annonce souvent que les méthodes de débordement sont plus performantes *mais* il y a les chaînages qui vont détèriorer le coefficient α.

- *Mémoire de débordement avec pointeurs*

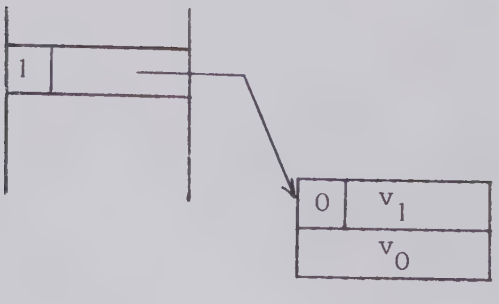
Il existe plusieurs façons de tenir compte des chaînages :

1. On laisse dans chaque mot de la mémoire un bit qui indique si le contenu de ce mot est soit une valeur soit une adresse dans le cas où il y a eu collision. Dans le cas d'une adresse, cette adresse est un pointeur vers un endroit de la mémoire où l'on a rangé les deux valeurs en collision. Dans le cas d'une collision d'ordre 3 on recommence avec le premier mot de la cellule contenant les deux valeurs et ainsi de suite.



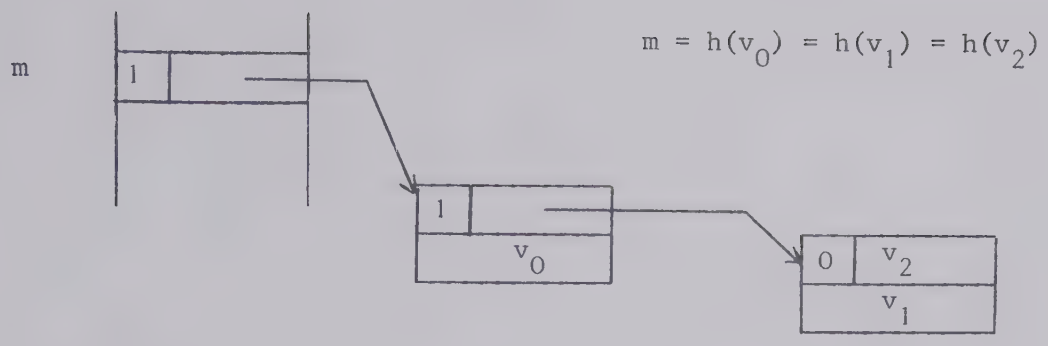
$m = h(v_0)$

Une valeur v_0



$m = h(v_0) = h(v_1)$

Deux valeurs v_0 et v_1



Trois valeurs v_1, v_2, v_3

Le nombre moyen d'accès β ne change pas beaucoup, on pourrait presque considérer qu'il diminue un peu en entrée sorties réelles car lorsqu'il y a collisions la dernière cellule qui contient deux valeurs fait certainement partie d'un même groupe et est accessible en un seul accès. (Cette réduction est d'ailleurs compressée pour les collisions d'ordre supérieur à 2 puisque le premier accès ne donne pas de valeur mais un pointeur.

En tenant compte des pointeurs, le nombre d'accès en moyenne est :

1 pour les collisions d'ordre 1
2,5 pour les collisions d'ordre 2
11/3 pour les collisions d'ordre 3

On vérifie aisément qu'il est de $P + 1 - \frac{1}{P}$ pour les collisions d'ordre $P > 1$. Et donc :

$$\beta = \frac{1}{X} \sum_{P=1}^{X} (P + 1 - \frac{1}{P})\ P\ C(P)$$

Soit $$\beta = \frac{2}{X} \sum_{P=1}^{X} \frac{(P + 1)\ P}{2} C(P) - \frac{1}{X} \sum_{P=1}^{X} C(P)$$

le premier terme a déjà été calculé

$$\frac{2}{X} \sum_{P=1}^{X} \frac{(P + 1)\ P}{2}\ C(P) = \frac{X(k + 2) - k}{X}$$

le deuxième peut s'écrire

$$J = \frac{1}{X} \sum_{P=1}^{X} C(P) = \frac{1}{X} \sum_{P=1}^{X} \frac{C_X^P\ (n - 1)^{X-P}}{X}$$

$$= \frac{1}{k_n^X}\ X\ n^X - \frac{1}{k} \left(\frac{n - 1}{n}\right)^X$$

Donc $$\beta = \frac{X(k + 2) - k}{X} - \frac{1}{k} \left(1 - \left(\frac{n - 1}{n}\right)^{k_n}\right)$$

Soit quand n est grand

$$\boxed{\beta \sim k + 2 - \frac{1}{k} (1 - e^{-k})}$$

Pour le calcul de α les collisions d'ordre 1 ne changent rien mais pour les collisions d'ordre supérieur, la taille de la mémoire de débordement *double* avec l'influence des pointeurs. La modification due au bit supplémentair est de l'ordre de $1/\text{Log}_2 [N]$ suivant la longueur de la représentation des valeurs de N.

$$M \sim n [k + 2e^{-k}]$$

et donc $$\boxed{\alpha = \frac{k}{k + 2e^{-k}} - \frac{1}{[\text{Log}_2 N]}}$$ (au lieu de $k/k + C^{-k}$)

Exemple

k = 1/2	α = 25 %	β = 1,72
k = 1	α = 55 %	β = 2,37
k = 2	α = 88 %	β = 3,57

Avec ce coefficient α corrigé,on constate que les performances deviennent bien inférieures à celles du meilleur "Hash Coding" HC3 puisqu'on avait pour α = 70 %, β = 1,68 alors qu'on a le même résultat ici pour α = 23 % environ.

2. On laisse toujours dans chaque mot de la mémoire principale un bit pour indiquer la présence d'une valeur ou d'un chaînage et on crée des cellules de 2, 3, 4, ... mots de la mémoire correspondant à l'ordre de la collision.

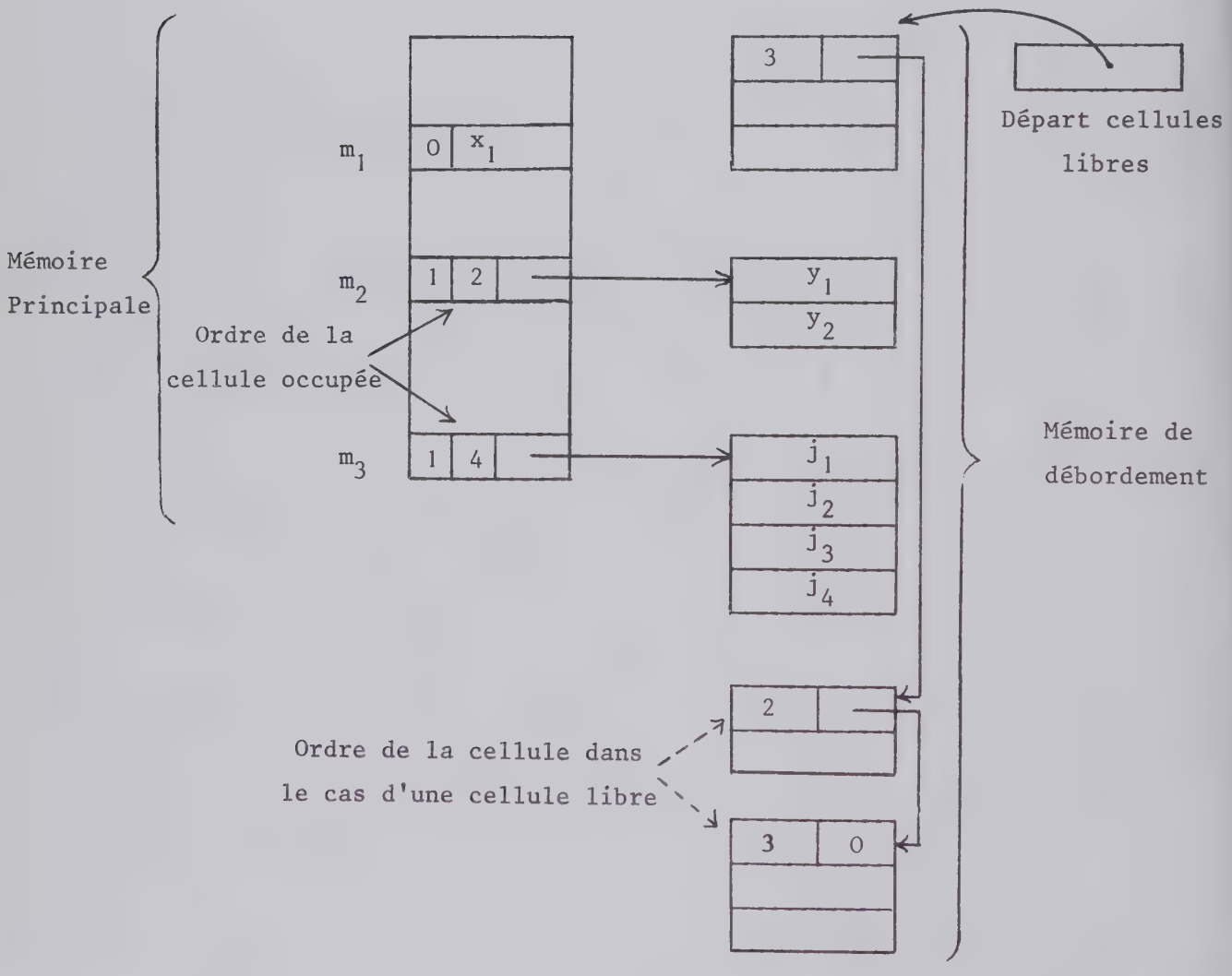
L'obstacle est que le rangement se fait au fur et à mesure et que la mémoire doit pouvoir évoluer. En fait, on peut résoudre ce problème avec une gestion dynamique des cellules de la mémoire de réserve.

Si une collision est d'ordre 3, on lui affecte une cellule de trois éléments et on prend dans la mémoire de débordement non encore occupée une cellule de 4 éléments dans laquelle on recopie l'ancienne cellule et la nouvelle valeur. On change ensuite le chaînage de départ.

La cellule ainsi libérée est chaînée avec la liste des cellules libres pour être réutilisée par exemple lors de la transformation d'une cellule de 2 éléments en cellule de trois éléments.

Il ne faut pas oublier de conserver la taille des cellules puisqu'elles sont variables. C'est très facile dans les cellules de réserve car les chaînages n'occupent que peu de place. Pour les cellules occupées, leur taille est conservée dans la mémoire principale avec le chaînage renvoyant vers la cellule. Il reste assez de place, N étant grand par rapport à X.

Il y aura très peu de cellules libres car l'ordre des collisions suit une loi sensiblement normale et il y a beaucoup plus de chance qu'il y ait création de cellules d'ordre 2 que d'ordre 3 par exemple. Les cellules libres seront rapidement réutilisées. Ceci se vérifie d'ailleurs très bien expérimentalement. On constate très souvent qu'il ne reste pas de cellules libres.



$$h(x_1) = m_1 \qquad (m_1) = 1$$

$$h(y_1) = h(y_2) = m_2 \qquad (m_2) = 2$$

$$h(j_1) = h(j_2) = h(j_3) = h(j_4) = m_3 \qquad (m_3) = 4$$

Calcul de α

Si on néglige la place occupée par les cellules libres qui ne peut être que très faible par rapport à la taille générale du Fichier. Seules, les collisions d'ordre 1 correspondent à des valeurs rangées dans la mémoire principale, les autres valeurs étant toutes dans la mémoire de débordement qui ne contient *aucun pointeur*.

Donc $$M = n + X - C(1)$$

Soit $$M \simeq n + k_n - k_n e^{-k}$$

et $$\boxed{\alpha \simeq \frac{k}{1 + k + ke^{-k}} - \frac{1}{\text{Log}_2 N}}$$

Calcul de β

Il faut cette fois $\frac{P + 1}{2} + 1$ recherches en moyenne pour les collisions d'ordre $P > 1$ et donc

$$\beta = \frac{1}{X} (1 \times C(1) + \sum_{P=2}^{X} (\frac{P + 3}{2})^P C(P))$$

Soit $$\beta = \frac{1}{X} \sum_{P=1}^{X} \frac{(P + 1)P}{2} C(P) + \frac{1}{X} \sum_{P=2}^{X} P \frac{C_X^P (n - 1)^{X-P}}{n^{X-1}}$$

Le premier terme a déjà été calculé, c'est le coefficient normal lorsqu'on ne tient pas compte des dommages. Le deuxième terme K peut s'écrire :

$$K = \sum_{P=1}^{X} \frac{P}{X} C_X^P \frac{(n - 1)^{X-P}}{n^X} - \frac{(n - 1)^{X-1}}{n^{X-1}}$$

Soit $$K = \sum_{q=0}^{X-1} C_{X-1}^{q} \; (n-1)^{(X-1)q} - \left(\frac{n-1}{n}\right)^{X-1}$$

$$K = 1 - \left(\frac{n-1}{n}\right)^{X-1}$$

et donc $$\beta = \frac{1}{2} \frac{X(k+2) - k}{X} + 1 - \left(\frac{n-1}{n}\right)^{X-1}$$

soit $$\boxed{\beta \simeq 2 + \frac{k}{2} - \frac{1}{C^k}}$$

Exemple

k = 1/2	α = 43 %	β = 1,64
k = 1	α = 71 %	β = 2,13
k = 2	α = 92 %	β = 2,87

Sans atteindre les résultats du Hash Coding HC3 cette méthode donne des résultats intéressants. Ils le sont surtout lorsqu'on introduit le facteur de groupage. A chaque fois que l'on fera appel à une valeur dans la mémoire de réserve, nous aurons un accès à la cellule ayant pour nombre d'éléments l'ordre de la collision en 1 accès réel. Nous aurons donc n'importe quelle valeur en *deux entrées sorties maximum*.

Le coefficient α ne change pas mais le coefficient β devient :

$$\beta = \frac{1}{X} \; (1 * C(1) + 2(X - C(1)))$$

Car il faut un accès pour les collisions d'ordre 1 et deux accès pour toutes les autres.

Soit $$\beta = 2 - \left(\frac{n-1}{n}\right)^{X-1}$$

soit $$\boxed{\beta \simeq 2 - e^{-k}}$$

Exemple

k = 1/2	α = 45 %	β = 1,39
k = 1	α = 71 %	β = 1,63
k = 2	α = 92 %	β = 1,87

De cette façon, cette méthode devient comparable à la meilleure méthode de Hash Coding HC3. Elle lui est légèrement inférieure mais présente l'avantage de donner une borne supérieure (2) au nombre d'accès réel.

Nous pouvons maintenant donner un tableau récapitulatif des méthodes de Hash Coding avec une mémoire de débordement.

Nous noterons :

MA	Méthode à chaînage simple
MB1	Méthode à cellules variables
MB1H	Méthode à cellules variables avec influence du facteur de groupage.

	MA	MB1	MB1M
α	$\frac{k}{k + 2e^{-k}}$	$\frac{k}{k + 1 + ke^{-k}}$	$\frac{k}{k + 1 + ke^{-k}}$
β	$k + 2 - \frac{k}{(1 - e^{-k})}$	$2 + \frac{k}{2} - e^{-k}$	$2 - e^{-k}$

c. Mémoires successives

Comme dans le "Hash Coding" usuel, on considère une suite de fonctions $(h_0, h_1, \ldots, h_i, \ldots)$. Au lieu de considérer que ces fonctions s'appliquent toutes dans une seule mémoire, on suppose une suite de mémoires $(M_0, M_1, \ldots, M_i, \ldots)$ disjointes et que chaque fonction $h_i(x)$ s'applique dans la mémoire $[M_i]$.

Les tailles des mémoires M_i sont notées $[M_i]$ et $M = \bigcup_i M_i$

$[M] = \sum [M_i]$

M est fini.

Nous poserons $[X] = k_0 [M_0]$.

k_0 est caractéristique du rapport entre la première mémoire et l'ensemble X des valeurs à ranger. Nous pouvons alors ranger les valeurs de X dans M de la façon suivante :

Par $h_0(x)$ on essaye de ranger la valeur x dans M_0. Si ce n'est pas possible (collision) on essaye dans M_1, si ce n'est pas possible dans M_2 etc...

Le nombre de valeurs qui se rangent dans M_0 est égal à

$[M_0] - C(0) = \alpha_0 [M_1] = \frac{\alpha_0}{k_0} [X]$.

α_1 peut être considéré comme le coefficient de remplissage de la première mémoire et on a $\alpha_0 = (1 - e^{-k_0})$

Il reste alors à ranger $X' = 1 - \frac{\alpha_0}{k_0} [X]$ valeurs dans une mémoire globale $M' = M - M_0$. Le problème de rangement se repose donc de la même façon pour les valeurs restantes dans la mémoire restante.

On peut donc définir de même $k_1 = [X'] / [M_1]$ et $\alpha_1 = (1 - e^{-k_1})$. On définit aussi le coefficient *global de remplissage de M'*.

$$\alpha' = [X'] / [M'] \quad \text{avec} \quad [M'] = [M] - [M_0]$$

Il vient donc

$$\beta = \frac{([X] - [X']) \times 1 + [X'] \ (1 + \beta')}{[X]} = 1 + (1 - \frac{\alpha_0}{k_0}) \ \beta' = 1 + \mu_0 \beta'$$

en posant $\mu_0 = 1 - \frac{\alpha_0}{k_0}$.

β' est le coefficient moyen de recherche dans la mémoire M'.

De même

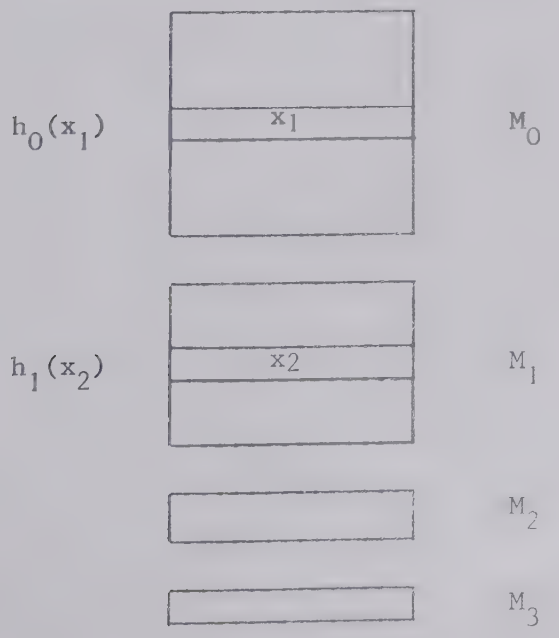
$$\gamma = (1 - \alpha_0) . \ 1 + \alpha_0 (1 + \gamma') = 1 + \alpha_0 \ \gamma'$$

Le problème est de trouver la succession des mémoires M_i pour avoir une utilisation optimale de la mémoire c'est-à-dire des coefficients β et γ minimaux.

Minimisation des coefficients β et γ

1. Problème pratique

Etant donnés la taille [M] de la mémoire disponible et le nombre [X] d'articles à ranger, déterminer le nombre et les tailles successives des mémoires M_i à réserver en vue de minimiser en moyenne β ou γ.



Il est nécessaire, pour le problème pratique que le nombre de mémoires soit fini et que ces mémoires soient de dimension entière. On peut donc limiter le nombre de mémoires dès que leur taille devient plus petite que 1. Il est possible qu'il existe des valeurs qui ne puissent se ranger dans la mémoire. L'expérience pratique montre que ce cas est très faible et que par exemple pour un fichier de 3 000 valeurs une zone de débordement supplémentaire de 3 ou 4 est satisfaisante.

Le problème pratique, qui demande une résolution arithmétique ne semble pas pouvoir être résolu directement. En effet, le nombre d'inconnues lui-même est une inconnue. Nous nous contenterons dans cette partie de donner une solution qui semble naturelle.

Le problème se reposant après chaque mémoire, on peut envisager la solution :

$$\alpha_i = \alpha = \text{constante}$$
$$k_i = -\text{Log}\ (1 - \alpha) = k = \text{constante}$$

Les mémoires ont alors leurs tailles en progression géométrique décroissante et connaissant $[M]$ il est alors aisé de déterminer leurs tailles ainsi que les coefficients β et γ.

Puisque toutes les mémoires ont un même coefficient de remplissage α, le coefficient de remplissage général est aussi α, et donc $\alpha = [X] / [M]$ ce qui permet de déduire k et la succession des mémoires.

Pour la première mémoire, on utilise une mémoire M_0 de taille $[M_0] = [X] / k$.

$[M_0] - C(0)$ valeurs sont rangées dans cette mémoire et il reste donc à ranger $[X] - [M_0] + C(0)$ valeurs et pour cela on utilise une mémoire M_1 de taille $[M_1] = ([X] - [M_0] + C(0)) / k$

soit $[M_1] = ([X] - \frac{[X]}{k} + \frac{[X]e^{-k}}{k}) / k$ et le coefficient de proportionalité entre les diverses mémoires est donc :

$$\boxed{a = 1 - \frac{1}{k} + \frac{e^{-k}}{k}}$$

et donc $$\beta = \frac{1}{k} \sum_{i=1}^{\infty} i(1 - \frac{1}{e^k})(1 - \frac{1}{k} + \frac{1}{ke^k})^i$$

Puisqu'il y a dans chaque mémoire $n' \times (1 - e^{-k})$ valeurs rangées d'accès i et $n' = (1 + \frac{1}{ke^k} - \frac{1}{k})[X]$.

Posons $1 - \frac{1}{k} + \frac{1}{ke^k} = a$ et calculons $I = \sum_{i=1}^{\infty} i\, a^i = a \sum_{i=1}^{\infty} (i - 1 + 1)a^{i-1}$

donc $I = a \sum_{i=1}^{\infty} (i - 1)a^{i-1} + a \sum_{i=1}^{\infty} a^{i-1} = a\, I + \frac{1}{i - a}$ soit $I = \frac{1}{(1 - a)^2}$

et $\beta = \frac{1}{k}(1 - \frac{1}{e^k}) \times \frac{1}{(\frac{1}{k} - \frac{1}{ke^k})^2} = k(\frac{e^k}{e^k - 1})$. Or $e^k = \frac{1}{1 - \alpha}$ et donc

$$\boxed{\beta = -\frac{\mathrm{Log}(1 - \alpha)}{\alpha}}$$

De même, on trouve

$$\boxed{\gamma = \frac{1}{1 - \alpha}}$$

On peut remarquer qu'on retrouve ici les résultats théoriques de W. W. PETERSON pour une mémoire sans débordement.

L'expérience vérifie parfaitement ces résultats et aboutit aux valeurs suivantes :

α	0,2	0,4	0,6	0,8	0,9
β	1,11	1,27	1,50	1,99	2,56
γ	1,25	1,67	2,50	5,00	10,00

On constate aisément que si β reste petit, le coefficient augmente sensiblement lorsque α devient voisin de 0,8. Il sera montré dans le paragraphe suivant qu'on *n'obtient jamais* le minimum de β et γ *pour les mêmes mémoires*. On aura donc intérêt à chercher un compromis *suivant l'utilisation* que l'on désire faire de la mémoire. Si on doit seulement vérifier que des valeurs sont dans la mémoire, on choisira des tailles successives telles que β soit minimum, par contre si très souvent les valeurs ne sont pas dans la mémoire on choisira γ minimum.

Ce résultat montrera aussi que ce n'est pas le moyen de création le plus économique qui donne le meilleur moyen d'utilisation.

2. *Problème théorique*

Soit S_u l'espace des suites de réels positifs u_j du segment $]0\ 1[$ tel que

$$\sum_{n=1}^{\infty} \prod_{j=1}^{n} u_j \quad \text{soit une série convergente}$$

Donnons l'énoncé des problèmes relatifs à β et γ.

Problème (β)

Soit $M = (\mu_0, \mu_1, \ldots, \mu_n, \ldots) \in S_\mu$

Trouver M de façon à minimiser

$$\beta = 1 + \sum_{n=1}^{\infty} \left(\prod_{i=1}^{n} \mu_i \right)$$

Compte tenu de la relation

$$\psi(\mu_0, \ldots, \mu_i, \ldots) = f(\alpha)$$

La contrainte ψ exprime que la réunion des M_i doit recouvrir la mémoire totale.

$$\psi(\mu_0, \mu_2, \ldots) = \frac{1}{k_0} + \frac{\mu_0}{k_1} + \ldots + \frac{\mu_0\ \mu_1\ \mu_{n-1}}{k_n} + \ldots = \frac{1}{\alpha}$$

k_0, k_1, k_n pouvant s'exprimer aisément à l'aide des μ_i :

$$\mu_i = 1 - \frac{\alpha_i}{k_i} \qquad k_i = -\mathrm{Log}(1 - \alpha_i)$$

Problème (γ)

Trouver $A = (\alpha_0, \ldots, \alpha_n, \ldots) \in S_\alpha$ de façon à minimiser

$$\gamma = 1 + \sum_{n=1}^{\infty} \left(\prod_{i=1}^{n} \alpha_i \right) \text{ compte tenu de}$$

$$\varphi(\alpha_0, \ldots, \alpha_n, \ldots) = g(\alpha)$$

La contrainte étant la même que pour le problème β au changement de paramètres près.

Les conditions posées et l'unicité de la solution (dérivées partielles positives) nous permet de donner un système équivalent dans le problème (β).

$$(S_\beta) \begin{cases} \dfrac{\partial \beta}{\partial \mu_i}(\mu_0, \ldots \mu_n, \ldots) = \left(\prod_{j=1}^{i-1} \mu_j \right) \times \beta(\mu_{i+1}, \mu_{i+2}, \ldots) \\ \dfrac{\partial \psi}{\partial \mu_i}(\mu_0, \ldots, \mu_n, \ldots) = \left(\prod_{j=1}^{i-1} \mu_j \right) \times \left[\psi(\mu_{i+1}, \mu_{i+2}, \ldots) + \dfrac{1}{1 - (i - k_i)e^{-k_i}} \right] \end{cases}$$

et d'une façon analogue un système équivalent pour (γ).

$$(S_\gamma) \begin{cases} \dfrac{\partial \gamma}{\partial \alpha_i}(\alpha_0, \alpha_2, \ldots) = \left(\prod_{j=1}^{i-1} \alpha_j\right) \times \beta(\alpha_{i+1}, \alpha_{i+2}, \ldots) \\ \dfrac{\partial \varphi}{\partial \alpha_i}(\alpha_0, \alpha_2, \ldots) = \left(\prod_{j=1}^{i-1} \mu_j\right) \times \left[\psi(\alpha_{i+1}, \alpha_{i+2}, \ldots) + \dfrac{1}{1 - (1 - \alpha_i)(1 + \mathrm{Log}(1 - \alpha_i))}\right] * \dfrac{\mathrm{Log}(1 - \alpha_i) + \dfrac{\alpha_i}{1 - i}}{\mathrm{Log}(1 - \alpha_i)} \end{cases}$$

Ces deux systèmes, quoique définissant les solutions cherchées, ne semblent pas pouvoir se résoudre par des méthodes classiques. Nous n'avons pas pu les résoudre directement mais ils nous permettent d'énoncer :

Proposition 1

La solution de (S_β) est la suite constante $\mu_i = 1 + \dfrac{\alpha}{Log(1 - \alpha)}$

C'est la solution étudiée dans le problème pratique. Nous avons seulement vérifié, grâce aux équations que c'était bien la solution donnant le minimum pour β.

En reportant cette solution dans (S_γ) nous pouvons énoncer :

Proposition 2

La solution $\alpha_i = \alpha$ = constante n'est pas solution de (S_γ)

Ce qui démontre l'incompatibilité des deux minimums.

Ne trouvant pas de méthode de résolution du système (S_γ) aboutissant à un résultat, l'essai de la plupart des solutions "intuitives" que nous avons essayé ayant échoué, nous avons cherché le moyen de trouver une solution approchée permettant de diminuer γ.

3. *Méthode de résolution approchée*

On utilise la relation $\gamma = 1 + \alpha_0 \gamma'$ et on remplace le problème possédant un nombre infini d'inconnues par un problème à deux inconnues.

Déterminer α_0 tel que γ soit le meilleur possible avec α_i constant pour $i \geqslant 1$.

C'est-à-dire qu'on se restreint à la méthode de décroissance géométrique à partir de la deuxième mémoire. On possède alors $\gamma = f(\alpha_0)$ et on cherche à minimiser cette fonction d'une variable sur $]0\ 1[$.

Une fois α_0 déterminé, on en déduit $M' = M - M_0$ et on recommence le procédé avec M' et α_1 puis avec W'' et α_2 etc...

Chaque itération du procédé fait diminuer γ. On s'arrête lorsque la taille de la mémoire M_i devient inférieur à 1.

La suite des $\gamma^{(n)}$ ainsi obtenue étant strictement décroissante et bornée inférieurement, tend évidemment vers une limite $\gamma^{(\infty)}$.

La question (non résolue) est de savoir : $\delta^{(\infty)}$ est-il égal à la valeur du minimum de γ ?

Expérimentalement, ces résultats sont confirmés par exemple :

γ	α	0,65
	Méthode à décroissance géométrique	2,86
	Méthode itérative	2,60

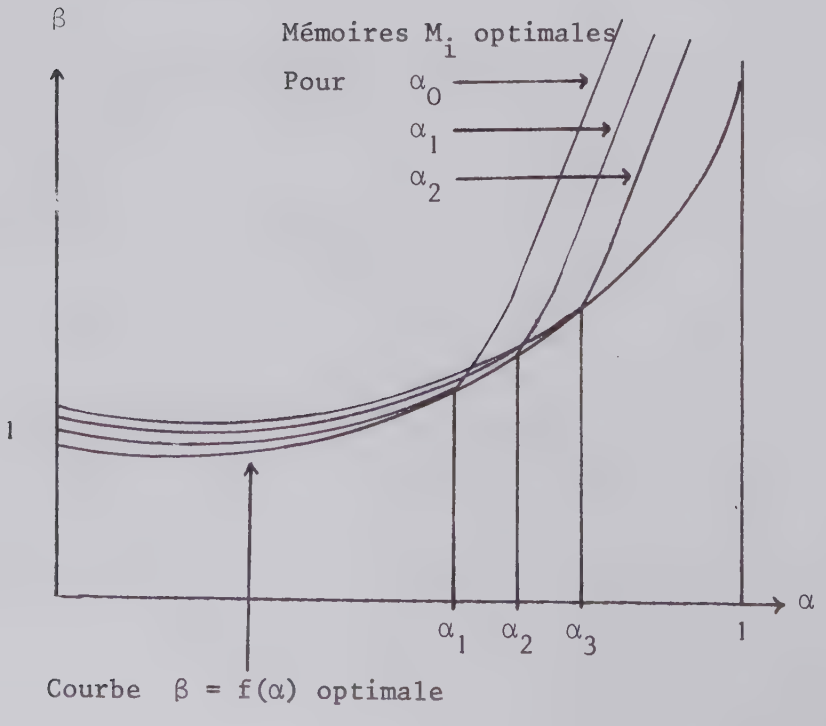
La différence devenant de plus en plus sensible lorsque α augmente, on constate d'autre part que l'augmentation correspondante de β reste faible et que cette méthode reste stable vis à vis d'un débordement supplémentaire.

Enfin, le nombre de mémoires à envisager est toujours plus faible que dans la méthode à décroissance géométrique.

Remarque

Le calcul d'optimalité de répartition des mémoires, donne une répartition idéale pour un *coefficient α choisi* puisque le coefficient de décroissance entre les mémoires dépend de k donc de α.

Si on étudie l'évolution des performances, par exemple de sur une répartition de mémoire optimale pour un α_0 choisi, on constate que jusqu'à ce que α soit égal à α_0 les performances ne sont pas loin des performances idéales pour le coefficient α en cours. Mais dès que α dépasse α_0 les performances se détèriorent rapidement et s'éloignent très vite de la position idéale.



De sorte que même si on ne sait pas trouver de répartition optimale sur γ pour un α donné on peut quand même définir pour une répartition donnée par expérience une valeur de γ qui semble optimale.

Ceci montre aussi que pour se prévenir de fonctions h_0 qui ne soient pas totalement satisfaisantes sur un fichier donné, on a intérêt à réduire un peu la taille de la première mémoire ce qui donne une optimalité pour un coefficient α supérieur mais permettra donc de ranger un peu plus de valeurs donc d'être plus sûr pour de petites variations.

d. Influence de l'adresse

Il est possible d'améliorer considérablement certains des résultats précédents en tenant compte de l'influence de l'adresse. Nous avons toujours utilisé dans nos évaluations de coefficient, le coefficient α habituellement utilisé. Nous avons vu dans le chapitre II que ce coefficient peut être supérieur à 1. En fait, dans aucune de nos méthodes sauf la méthode directe nous n'avons rencontré ce cas. Nous avons vu ainsi qu'une meilleure définition du coefficient α est réalisée en donnant comme quantité minimale de mémoire nécessaire $[X] \text{ Log } [N] / [X]$, formule qui peut s'interpréter en exprimant que l'adresse est une partie effective de l'information.

La méthode directe d'ailleurs utilise ce fait puisque les valeurs sont considérées directement comme des adresses (c'est une fonction de Hash Coding particulière et idéale).

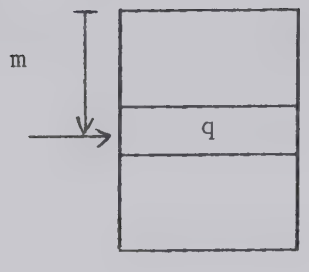
Il est possible d'améliorer les méthodes de Hash Coding avec débordement et la méthode des mémoires successives lorsqu'on utilise comme fonction unique de Hash Coding h_0 : le reste de la division par la taille de la mémoire principale. Cette fonction, nous l'avons vue, est la meilleure.

Il n'est pas utile dans le cas de cette fonction de conserver la valeur elle-même, il suffit de *conserver le quotient* de la division.

$x = [M_1] * q + m$

m adresse

q quotient



L'adresse d'une mémoire m et son contenu q permettent bien entendu de retrouver la valeur x. Ceci peut poser des problèmes avec le quotient nul car la valeur nulle indique l'absence de présence d'une valeur, il suffit de ranger q + 1.

Dans la méthode de débordement avec chaînage, on ne range que les quotients dans les cellules de débordement puisque l'adresse du point d'entrée dans la sous liste fournit le reste.

Dans la méthode des mémoires successives, lorsqu'il y a collision, on range le nouveau quotient dans la mémoire suivante.

S'il fallait $\text{Log}_2 [N]$ bits pour conserver une valeur x, avec cette remarque il suffit de $\text{Log}_2 [N] - \text{Log}_2 [M_i]$ bits pour conserver n'importe quel quotient d'une valeur de x par la taille de la mémoire M_i. On retrouve ici pratiquement l'idée intuitive qui nous avait permis de retrouver la formule $[X] \text{ Log } \frac{[N]}{[X]}$ concernant la mémoire minimale nécessaire a priori.

Cette amélioration ne peut se réaliser avec la méthode de Hash Coding normale, sans débordement car on ne sait jamais si une valeur de la mémoire est un point d'entrée direct (obtenu par h_0) ou un point obtenu indirectement (par un $h_i(x)$). De plus on ne peut, connaissant uniquement un quotient dans une mémoire, retrouver le point d'entrée d'où il provient.

Les coefficients de performance se trouvent sensiblement améliorés du fait de cette économie de mémoire, à l'avantage du coefficient α.

1. Débordement avec chaînages

On retrouve le même gain dans tous les mots de la mémoire ce qui fait que si $\mathcal{M}$ était la quantité de mémoire utilisée, la mémoire réellement nécessaire est $\mathcal{M} * \frac{\text{Log } ([N] / [M_0])}{\text{Log}[N]}$

Ceci fait que le coefficient α est augmenté dans le rapport inverse :

$$e = \frac{\text{Log } [N]}{\text{Log } ([N] / [M_0]} = \frac{\text{Log}[N]}{\text{Log } (k\frac{[N]}{X})}$$

et donc pour les trois méthodes de débordement, on trouve les formules :

	MA	MB1	MB1M
α	$\frac{k_e}{k + 2e^{-k}}$	$\frac{k_e}{k + 1 + k\, e^{-k}}$	$\frac{k_e}{k + 1 + k\, e^{-k}}$
β	$k + 2\,\frac{k}{1 - e^{-k}}$	$2 - e^{-k} + \frac{k}{2}$	$2 - e^{-k}$

Ce qui donne donc par exemple pour la méthode MB1M :

1. Si on suppose $[X] = 2^{18} \simeq 25\ 000$ et $[N] = 2^{32} \simeq 4 \times 10^9$

k = 1/2	$\alpha = 1,12$	$\beta = 1,39$
k = 1	$\alpha = 1,63$	$\beta = 1,63$
k = 2	$\alpha = 1,93$	$\beta = 1,87$

2. Si on suppose $[X] = 2^{10} \simeq 1\ 000$ et $[N] = 2^{24} \simeq 8\ 000\ 000$

k = 1/2	$\alpha = 0,81$	$\beta = 1,39$
k = 1	$\alpha = 1,20$	$\beta = 1,63$
k = 2	$\alpha = 1,47$	$\beta = 1,87$

De ceci, on peut faire plusieurs remarques très importantes :

1. Les résultats sont extraordinairement bons et ne peuvent même plus être réellement comparés à ceux du Hash Coding Normal puisque *α peut être largement supérieur à 1*.

On peut interpréter la ligne k = 2 $\alpha = 1,93$ $\beta = 1,87$ de la façon suivante :

Il est possible de ranger 250 000 informations de 32 bits dans un peu plus de 125 000 mots de 32 bits (environ la moitié de la taille du fichier) en pouvant retrouver n'importe quelle valeur avec 1,87 accès en moyenne.

Ce résultat bouscule certaines idées reçues en informatique, en particulier celle qui indiquait que le Hash Coding est une bonne méthode permettant de retrouver rapidement une information mais qui malheureusement (sic) prend beaucoup de place !!!

3. Le nombre moyen d'accès β ne dépend plus uniquement de α. Il dépend aussi de la taille du fichier.

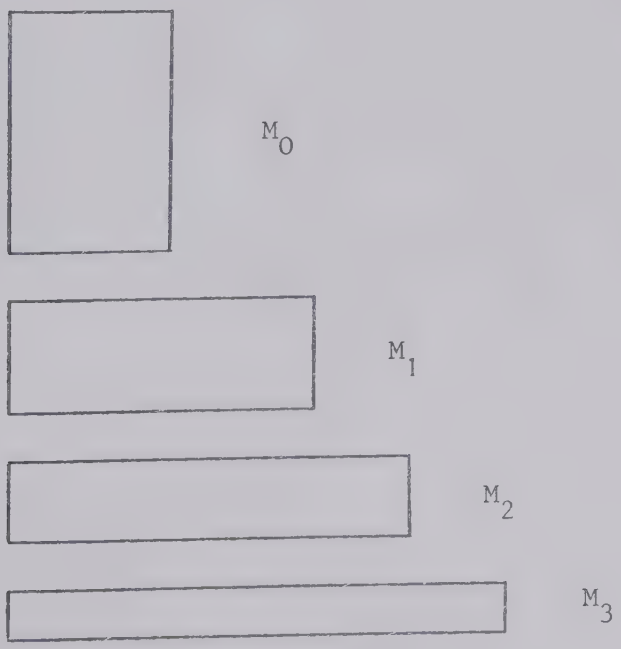
Les deux exemples donnés pour MBIM laissent entrevoir que pour un rapport constant entre N et X (même proportion de valeurs dans X par rapport à N) et pour un même coefficient α, *il faut moins d'accès pour retrouver une valeur dans un gros fichier que dans un petit*.

On retrouve plus rapidement une valeur sur 250 000 parmi 4 milliards qu'une valeur sur 1 000 parmi 8 millions !!!

2. *Mémoires successives*

Les différentes mémoires de la méthode à mémoires successives sont de tailles différentes et de longueurs de mots différentes. La première, la plus grande possèdera des mots très courts, la deuxième, un peu plus petite, des mots plus grands et ainsi de suite...

Les calculs d'optimalité que nous avons faits ne sont bien sûr plus valables compte tenu de cette réduction. Nous n'avons pas pu trouver de méthode permettant de découvrir une répartition optimale des mémoires M_i.



Si on effectue des calculs sur les répartitions optimales de la méthode normale, on obtient, après réduction les valeurs suivantes pour β et γ :

α	0,33	0,62	0,87	1,13	1,25
β	1,11	1,27	1,50	1,99	2,56
γ	1,25	1,67	2,50	5,00	10,00

Là aussi, les résultats sont excellents et on peut faire les mêmes remarques que pour les méthodes avec débordement.

De plus, lorsqu'on fixe les mémoires successives, la méthode est optimale pour un certain coefficient α qui peut se déterminer expérimentalement mais la mémoire n'est *pas encore "pleine"*.

Il est possible d'ajouter de nouvelles valeurs. Bien sûr, si aucune zone de débordement supplémentaire n'est autorisée, un certain nombre de valeurs peuvent être refusées c'est-à-dire qu'elles ne peuvent pas être insérées dans la mémoire alors que d'autres le pourraient. Est-ce-une gène ? Dans la méthode séquentielle, lorsqu'il n'y a plus de place : la mémoire est "pleine" et on refuse toutes les valeurs supplémentaires alors que dans cette méthode, on commence à refuser des valeurs lorsqu'on en a mis déjà environ 1,5 fois plus.

6. *Méthode approximative*

La méthode que nous nommons "Approximative" pourrait être considérée comme étant une méthode de Hash Coding particulière car elle utilise aussi la notion de transformation clef → adresse. Elle en diffère cependant par deux aspects :

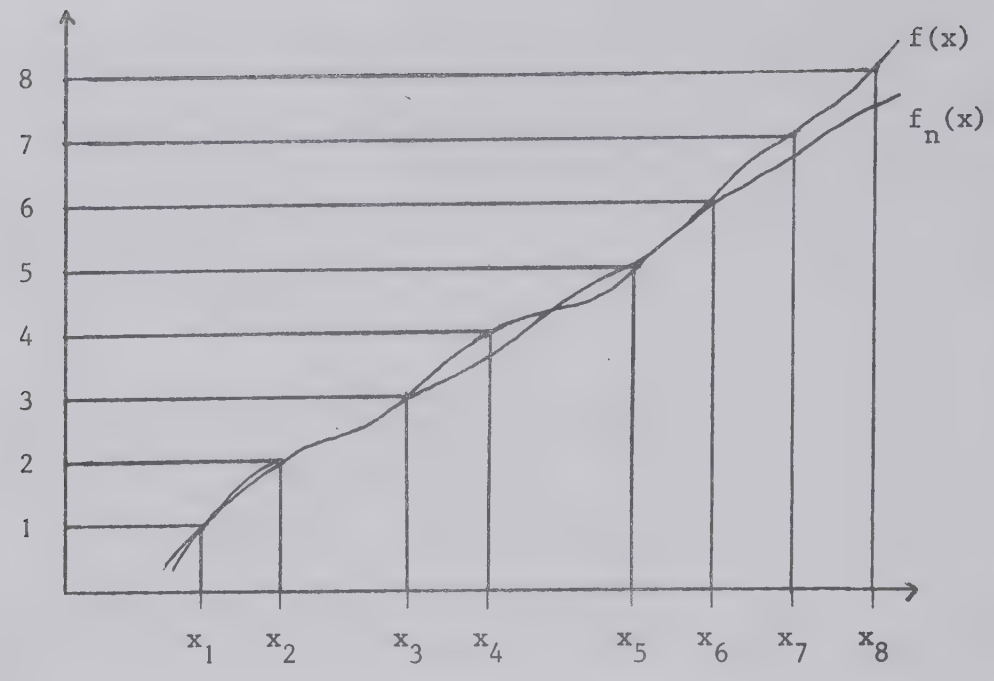
1. Elle est bâtie sur le caractère local des informations, on ne cherche pas à obtenir directement une adresse où se trouve la valeur mais un endroit où se trouve approximativement la valeur.

2. Elle permet des utilisations différentes de celles du Hash Coding, en particulier parce que le fichier reste ordonné dans la mémoire.

Méthode

Supposons connu un ordre sur l'ensemble N. Et supposons l'ensemble des valeurs X rangé en ordre croissant (c'est ainsi que débute la méthode dichotomique). Considérons la fonction f(x) qui, à chaque valeur de X fait correspondre son numéro d'ordre dans la mémoire. Cette fonction f(x) est une transformation clef → adresse idéale, elle fait correspondre à chaque valeur son adresse directement. En fait, cette fonction est donnée par l'ensemble des valeurs qu'elle prend sur l'ensemble X et nous ne sommes pas plus avancés qu'avant. Connaître une fonction par sa valeur en chaque point ne présente pas d'intérêt pour notre problème.

Il serait plus intéressant de connaître une fonction $f_n(x)$ qui approche f(x) suffisamment près et qui soit aisément calculable. La méthode approximative consiste à chercher une telle fonction f_n.



Connaissant $f_n(x)$, la donnée de x permettra ensuite de déterminer approximativement où se trouve x dans la mémoire.

Nous conserverons $f_n(x)$ sous forme d'échantillons et utilisant un *procédé d'interpolation local*, il est possible, étant donnée une valeur x, de calculer la valeur $f_n(x)$ correspondante.

La méthode locale d'interpolation est décrite dans l'annexe C. Son principe réside en une corrélation entre l'ensemble des échantillons et une fonction à support compact qui peut être discontinue. Le support compact et réduit de ces petites fonctions d'interpolation, donne un caractère essentiellement local puisque seuls les échantillons proches de la valeur à déterminer interviennent. Les fonctions d'interpolation sont constituées d'arcs de polynômes dont le degré n conditionne la précision du procédé d'interpolation.

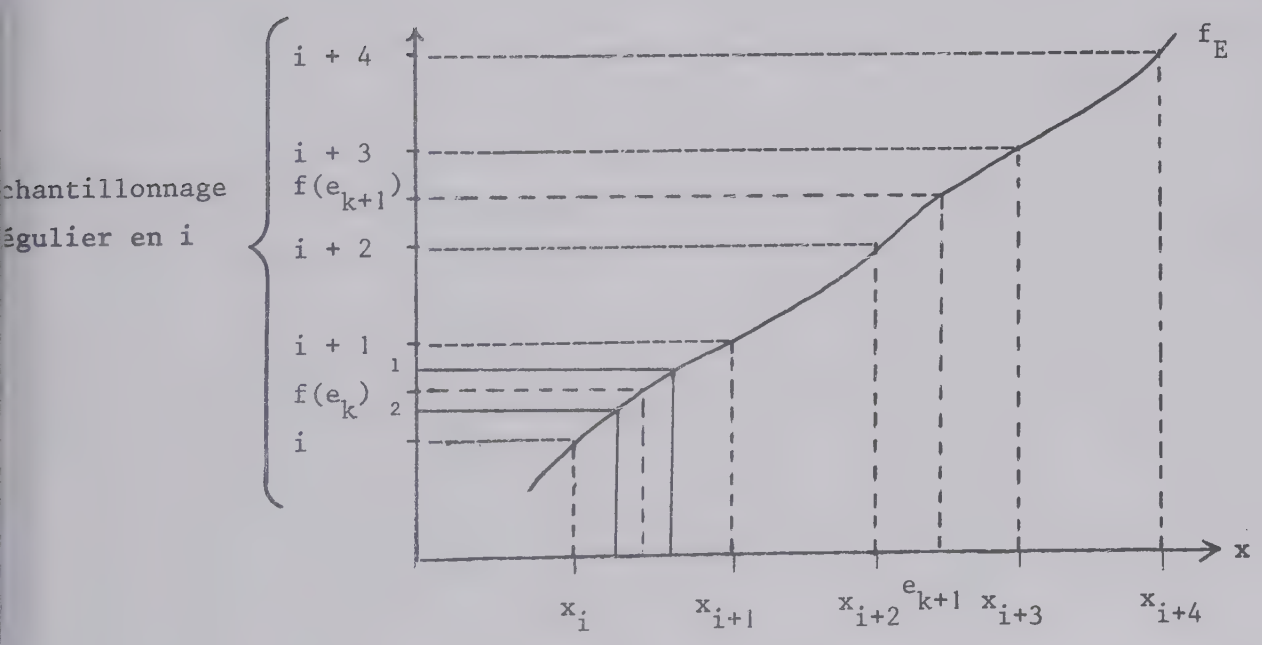
Les échantillons intervenant dans ce procédé doivent être régulièrement espacés, nous dirons que le pas d'échantillonnage doit être régulier.

Notre première étape va donc être de déterminer un échantillonnage $\{e_i\}_E$ de la fonction $f_n(x)$. Nous n'allons pas utiliser pour cela l'ensemble des valeurs de X car il faudrait attendre de toutes les avoir avant de commencer l'organisation. Nous supposerons qu'un sous ensemble X_E de X composé des premières valeurs donne, une fois ordonné, une allure représentative à un facteur d'échelle près de la courbe de rangement f.

a. Obtention de l'échantillonnage de $f_n(x)$

On commence par ordonner l'ensemble des valeurs de X_E. Il ne suffit pas ensuite de sélectionner certaines valeurs de X_E et de les conserver comme échantillons car il y a peu de chances que ces valeurs réalisent un échantillonnage à pas régulier et nous avons vu que c'est indispensable pour notre procédé d'interpolation.

Par contre, la courbe f_E obtenue est actuellement à pas régulier suivant les adresses et considérant ce pas régulier, il est possible de calculer la valeur en n'importe quel point de f_E.



Echantillonnage régulier en e_k à déterminer.

On décide alors d'un pas d'échantillonnage régulier sur les valeurs de x et on choisit un échantillonnage $\{e_k\}$ sans s'occuper de savoir si ces valeurs correspondent à des valeurs de x ou pas. On cherche ensuite les valeurs des échantillons $f_E(e_k)$ correspondants. Pour cela, on utilise le procédé dichotomique suivant :

Supposons le compris entre x_i et x_{i+1}. On calcule par interpolation la valeur de $f_E^{-1}(\frac{i + (i + 1)}{2})$, cela est possible puisque l'échantillonnage est régulier suivant les adresses et que les valeurs de f_{i-1}, f_i, f_{i+1}, f_{i+2}, ... sont connues. On compare ensuite la valeur de $f_E^{-i}(\frac{i + (i + 1)}{2})$ à e_k ce qui détermine où se trouve $f_E(e_k)$, soit dans l'intervalle $[i, \frac{i + (i + 1)}{2}]$ ou $[\frac{i + (i + 1)}{2}\ i + 1]$ et on continue de même avec la valeur du milieu de cet intervalle. On peut arrêter le procédé lorsque la valeur $f_E(e_k)$ calculée est connue approximativement à une adresse près sur l'ensemble du fichier X.

On a donc ainsi défini une suite de valeurs d'échantillonnage $\{e_k\}$ qui ne sont certainement pas dans X avec une correspondance $\{f_E(e_k)\}$ valeurs qui ne sont pas des adresses réelles. On modifie ensuite par homothéties, l'ensemble des valeurs $\{f_E(e_k)\}$ à un ensemble $\{f_n(e_k)\}$ suivant le rapport entre le nombre d'éléments de X_E et celui de X.

b. Méthode de rangement

Une fois obtenu un échantillonnage régulier suivant x de la courbe $f_n(x)$ on peut commencer à ranger les valeurs (les valeurs de X_E et les suivantes).

Pour chaque valeur x, on calcule par interpolation la valeur de $f_n(x_1)$ correspondante. On considère le nombre entier le plus proche de cette valeur comme étant une adresse et on range la valeur x_1 à cette adresse. S'il y a collision, on déplace *localement* les valeurs déjà rangées autour de cette adresse pour maintenir le fichier ordonné en mémoire.

Tant que la mémoire n'est pas trop pleine (ici aussi intervient le coefficient α) on constate que les déplacements ne sont pas nombreux et qu'il y a suffisamment de "trous" pour que le nombre de valeurs à déplacer soit à chaque fois très faible.

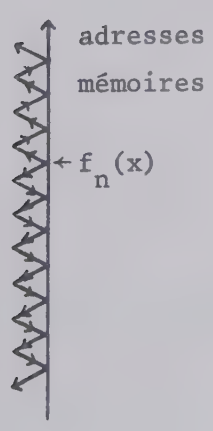
Là aussi, comme pour le Hash Coding, on constate que cette méthode est performante pour le coefficient d'occupation $\alpha \leqslant 0,7$ mais que les performances se dégradent très vite au-dessus.

c. Méthodes de recherche

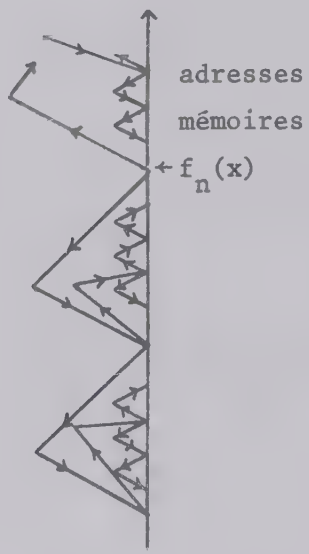
Pour rechercher une valeur dans la mémoire ou pour démontrer son absence, on peut alors utiliser deux procédés :

1. Il est possible comme pour la construction de la mémoire d'effectuer une recherche séquentielle locale à partir de l'adresse $f_n(x)$ trouvée. Par comparaison, entre x et la valeur contenue dans l'adresse $f_n(x)$ on connaît le sens vers lequel on doit poursuivre la recherche.

2. Les valeurs étant ordonnées et connaissant une marge de l'erreur que l'on commet par interpolation (cette erreur peut être précisée au fur et à mesure de la construction) il est possible d'effectuer une recherche dichotomique locale à partir de l'adresse $f_n(x)$ calculée



Recherche séquentielle locale

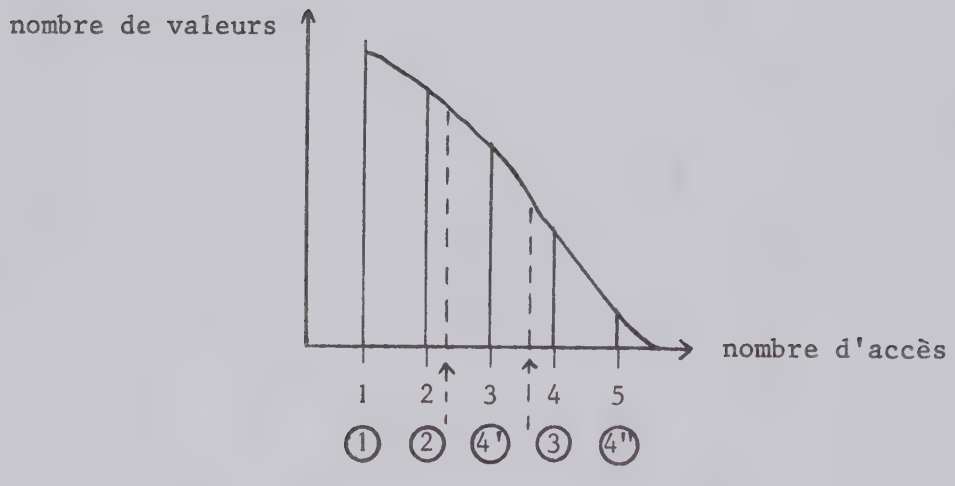


Recherche dichotomique locale

Si on sait que l'on trouve par $f_n(x)$ à une adresse qui est en moyenne à 4 près, on peut rechercher en $f_n(x)$ puis en $f_n(x) + 4$ et surtout le cas en $f_n(x) + 2$ ou $f_n(x) + 6$ etc...

Une manière de construire la meilleure recherche dichotomique locale serait, au fur et à mesure de l'établissement de la mémoire de déterminer le nombre de valeurs accessibles en 1 accès, 2 accès, 3 accès... par la méthode de recherche séquentielle locale.

On peut ensuite tracer l'histogramme de ces nombres d'accès :



et déterminer par des évaluations de surface l'ordre dans lequel il faut effectuer les tests.

d. Les résultats

Des essais ont été effectués avec un ensemble de nombres aléatoires sur CII 10070. La fonction donnant les nombres est la suivante :

$$x_{i+1} = (2^{10} + 1)x^i + 101 \qquad \text{avec} \quad x_0 = 1$$

La fonction d'interpolation est la fonction $g_1(x)$ (voir Annexe C). E est le pourcentage d'échantillons par rapport à l'ensemble de valeurs X à ranger.

X = 10 %

$\alpha = 0,6$ $\begin{cases} \beta = 1,71 \\ \gamma = 2,64 \end{cases}$

$\alpha = 0,7$ $\begin{cases} \beta = 2,19 \\ \gamma = 3,99 \end{cases}$

X = 40 %

$\alpha = 0,7$ $\begin{cases} \beta = 1,34 \\ \gamma = 1,67 \end{cases}$

La recherche locale est une recherche séquentielle. Des résultats plus complets sont donnés dans l'annexe B.

La répartition des valeurs de X étant régulière, nous avons pensé que l'allure générale de la courbe de rangement était une droite et nous avons alors pensé à modifier ces valeurs de telle façon que la courbe n'ait plus une allure rectiligne mais telle qu'il y ait une concentration de valeurs.

Chaque valeur y est telle que :

$$y = a(1 + (\frac{3x}{a - 1})^3)$$

On constate bien qu'il n'y a que très peu de changements.

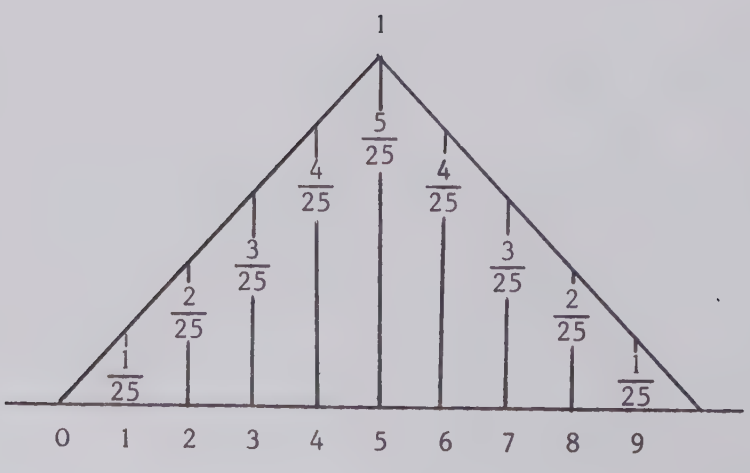
$E = 10\ \%$ $\qquad$ $\alpha = 0,7$ $\qquad$ $\begin{cases} \beta = 2,08 \\ \gamma = 3,76 \end{cases}$

e. Lissage de $f_n(x)$

Pour un nombre d'échantillons assez faible, nous avons pensé que quelques échantillons particuliers pouvaient avoir une importance désagréable pour notre procédé. Nous avons donc essayé d'améliorer ces résultats en effectuant un lissage de la courbe primitive obtenue à partir des échantillons.

L'échantillonnage varie entre $2\ \% \leqslant E \leqslant 8\ \%$.

Le filtre utilisé est un filtre à 9 valeurs.



Les coefficients sont :

$$[\frac{1}{25}, \frac{2}{25}, \frac{3}{25}, \frac{4}{25}, \frac{5}{25}, \frac{4}{25}, \frac{3}{25}, \frac{2}{25}, \frac{1}{25}]$$

Les résultats sont les suivants : $\alpha = 0,7$

	E = 4 %		E = 6 %		E = 8 %	
	β	γ	β	γ	β	γ
sans filtre	7,36	15,1	5,18	10,26	5,04	9,31
avec filtre	6,53	10 55	5,23	7,91	4,66	7,08

On constate bien une amélioration due au lissage. On remarquera (Annexe B) que cette amélioration est conséquente pour α compris entre 0,5 et 0,9.

f. Conclusion

La méthode approximative semble moins bonne que les méthodes de Hash Coding développées dans le chapitre 5. Elle peut néanmois être améliorée de plusieurs façons, d'une part en affinant la méthode de lissage, d'autre part en calculant effectivement les coefficients d'une méthode de recherche dichotomique locale.

Ses coefficients β et γ sont tout de même excellents et cette méthode permet de plus d'utiliser la mémoire sous la forme suivante :

1. Quelle est la valeur x la plus proche d'une valeur x_0 donnée ?

2. Quelles sont toutes les valeurs de la mémoire comprises entre x_1 et x_2 ?

Ce qui est bien sûr impossible sinon très lourd avec les organisations de Hash Coding puisque dans ces dernières, les valeurs sont dipersées et non en ordre croissant comme dans la méthode approximative.

De plus cette méthode s'adapte très bien lorsque la mémoire est une mémoire externe et qu'on accède à un groupe d'informations en une seule entrée sortie. En effet, lorsqu'on lit le groupe de valeurs correspondant à l'adresse calculée, on accède à tout un voisinage de cette adresse dans lequel la valeur a de fortes chances de se trouver.

ANNEXE A

ANNEXE A

Méthode approximative appliquée aux Tris

La méthode approximative donnant pour produit un fichier final ordonné, nous comparons dans cette annexe ses performances relativement aux performances habituelles des Tris. Nous comparerons principalement avec une méthode de tri considérée comme une des meilleures : le tri par interclassement (par monotonies).

1. Tri par interclassement

Nous décrirons ce tri sur un exemple :

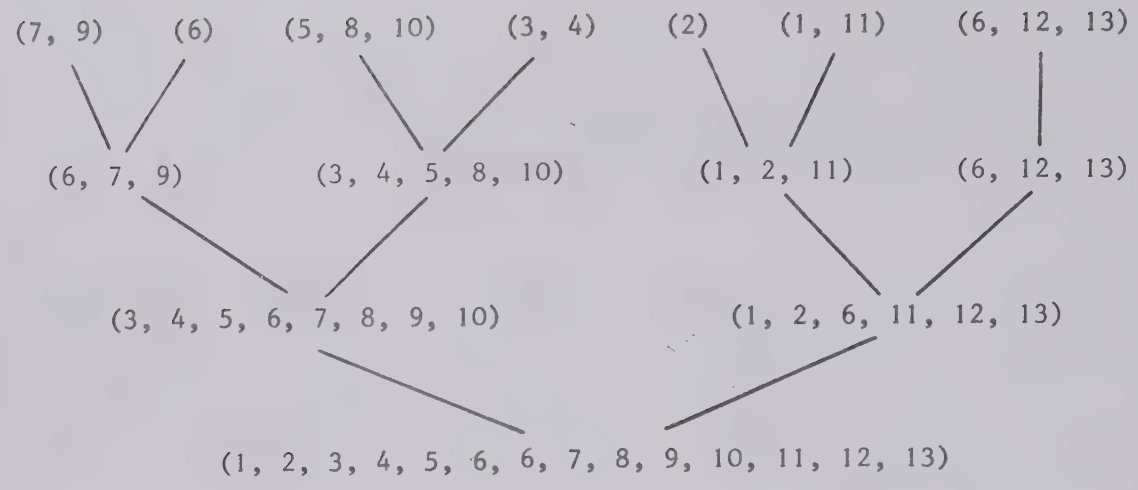
Soit à trier le fichier suivant par ordre croissant :

7, 9, 6, 5, 8, 10, 3, 4, 2, 1, 11, 6, 12, 13

On définit des monotonies comme étant des sous suites d'éléments triées dans l'ordre désiré. On sépare alors le fichier en monotonies maximales c'est-à-dire des monotonies qui ne peuvent être incluses dans aucune autre.

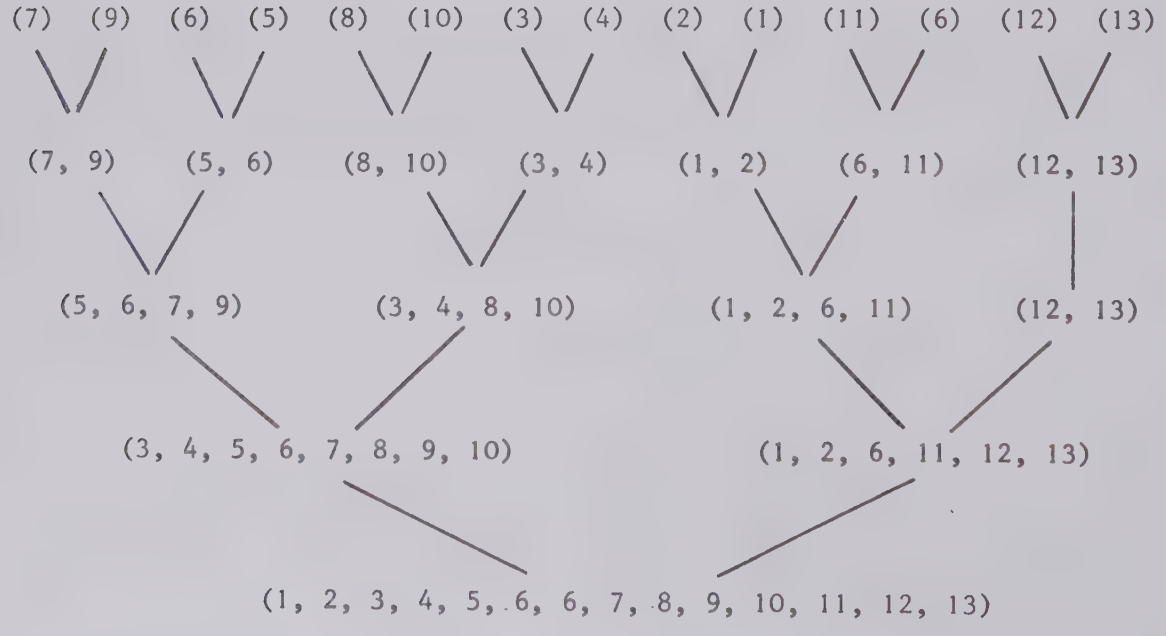
(7, 9), (6), (5, 8, 10), (3, 4), (2), (1, 11), (6, 12, 13)

On interclasse alors deux par deux les monotonies composant le fichier initial. On diminue alors de moitié le nombre de monotonies. On itère le procédé jusqu'à ce qu'il ne reste plus qu'une seule monotonie.



Le fichier est alors trié.

Une première méthode dite aveugle procède comme si les monotonies étaient composées d'un élément lors du premier traitement. Pour cette méthode, les monotonies sont de longueur quasiment fixe et le temps de traitement est donc fixe. Elle est aussi plus simple à programmer.



2. *Caractéristiques des tris sur mémoires externes*

Il n'existe pas de méthode spécifique au tri de Fichiers rangés sur disque. **Aussi**, avons nous adapté la méthode de tri par interclassement aveugle prévue sur bande pour le tri sur disque.

a. Principe

Définissons deux termes : fusion et distribution.

Fusion

Cela consiste à partir de deux **ou** plusieurs monotonies, une seule monotonie contenant tous les éléments des monotonies origines.

Les éléments à fusionner ne peuvent venir directement, au fur et à mesure des besoins, des bandes magnétiques. Les informations sont groupées en blocs d'enregistrement qui sont lus en une seule entrée-sortie. Ceci nécessite une ou plusieurs mémoires tampon (ou buffer) en mémoire centrale.

Distribution

Ceci consiste à ranger sur bande les monotonies produites par le tri interne. Il existe deux problèmes :

- Le résultat doit être stocké en zone intermédiaire (buffer) pour être vidé sur la mémoire externe par blocs.

- Sur quel dérouleur écrire cette zone après le remplissage et le tri du buffer ?

b. Application au tri sur disque

On se place dans le cas d'une seule unité de disques. On retrouve les mêmes étapes que pour le tri sur bande.

Distribution

Lecture du fichier bloc par bloc et écriture sur une zone tampon. On effectue le tri de ces zones en mémoire centrale et on récrit ce bloc sur disque.

Fusion

Il existe plusieurs variantes. Ne tenant compte que du nombre d'entrées-sorties, le tri sur disque qui nous a servi à tester cette méthode pour effectuer une comparaison avec la méthode approximative utilise un rangement séquentiel des blocs.

Le fichier à trier doit être conservé (hypothèse de travail), il se trouve sur disque dans la zone 1 ; le tri se réalise dans une zone 2 que l'on sépare en zone A et zone B.

Longueur de zone 2 = 2 fois longueur de zone 1
Longueur de zone 1 = longueur de zone A = longueur de zone B

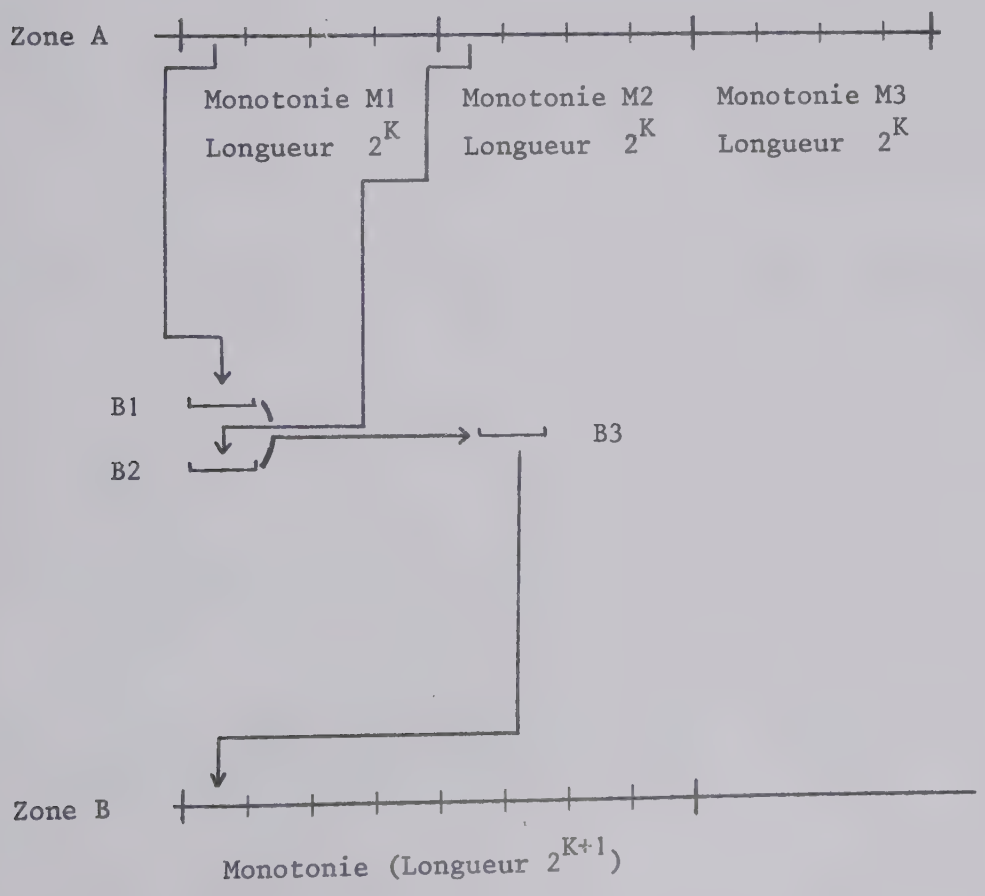
- *Lecture du fichier entrée bloc par bloc*

Chaque bloc trié en mémoire centrale est rangé séquentiellement dans la zone A. A la fin de cette étape, la zone A comprend un ensemble de blocs triés représentant le fichier entrée.

- *Fusion de ces blocs*

Nous utilisons trois buffers B1, B2, B3 de même longueur. Au départ de chaque étape de la fusion, nous possédons dans A (ou dans B) des monotonies constituées d'un ou plusieurs blocs, ce nombre de blocs par monotonie étant fixe pour une étape donnée et égal à 2^K (K est le numéro de l'étape).

On fusionne deux monotonies M1 et M2 : lecture du premier bloc de M1 et de M2 dans B1 et B2. On fusionne B1 et B2 à l'aide de B3. Lorsque B3 est plein, il est vidé sur la zone B. Si B1 (ou B2) devient vide, il est rempli avec le bloc suivant de la monotonie, il suffit de recopier l'autre dans B et la fusion de M1 et M2 est terminée.



c. Résultat du tri-fusion sur disque

Résultats théoriques

Soit X le nombre d'articles au total.

La place occupée sur disque est 2X. La place occupée en mémoire centrale est 3S, S étant le nombre d'articles contenus dans un buffer.

Pour le premier tri, on lit une fois et on écrit une fois chaque bloc. Sachant qu'il y a X/S blocs, on a 2X/S entrées-sorties.

Pour chaque étape de la fusion, on lit et on écrit une fois chaque bloc donc de même 2X/S entrées/sorties.

Si on a L blocs, il faut un nombre d'étapes K tel que $2^{K-1} < L \leqslant 2^{K}$.

Donc ici, nous avons $K \geqslant \text{Log}_2$ X/S, il faut donc pour réaliser le tri

2.(K + 1)/S entrées-sorties

Résultats expérimentaux

Ces résultats ont été obtenus à l'aide d'un fichier de 2 000 nombres aléatoires générés par une fonction standard de la CDC 3600.

S	20	30	40	50	100	200	300	400	667
Nb d'E/S	1 600	1 072	700	560	240	100	56	40	18

3. *Méthode approximative*

La méthode approximative a déjà été décrite et de nombreux résultats sont donnés dans l'annexe B. Elle de décompose en plusieurs étapes :

- Détermination de la fonction : adresse = f(contenu)

Nous considérons qu'un certain nombre des premières valeurs donne une allure approximative de la courbe de rangement de tout le fichier. On choisit donc les M premières valeurs qui nous servent d'échantillons. La méthode d'interpolation choisie est décrite dans l'annexe C.

- Rangement

Si l'adresse calculée est occupée on cherche la place où insérer la valeur à ranger.

a. Résultats expérimentaux

Avec la fonction d'interpolation g_1 et en considérant qu'il n'y a qu'une seule valeur par buffer (S = 1) on obtient :

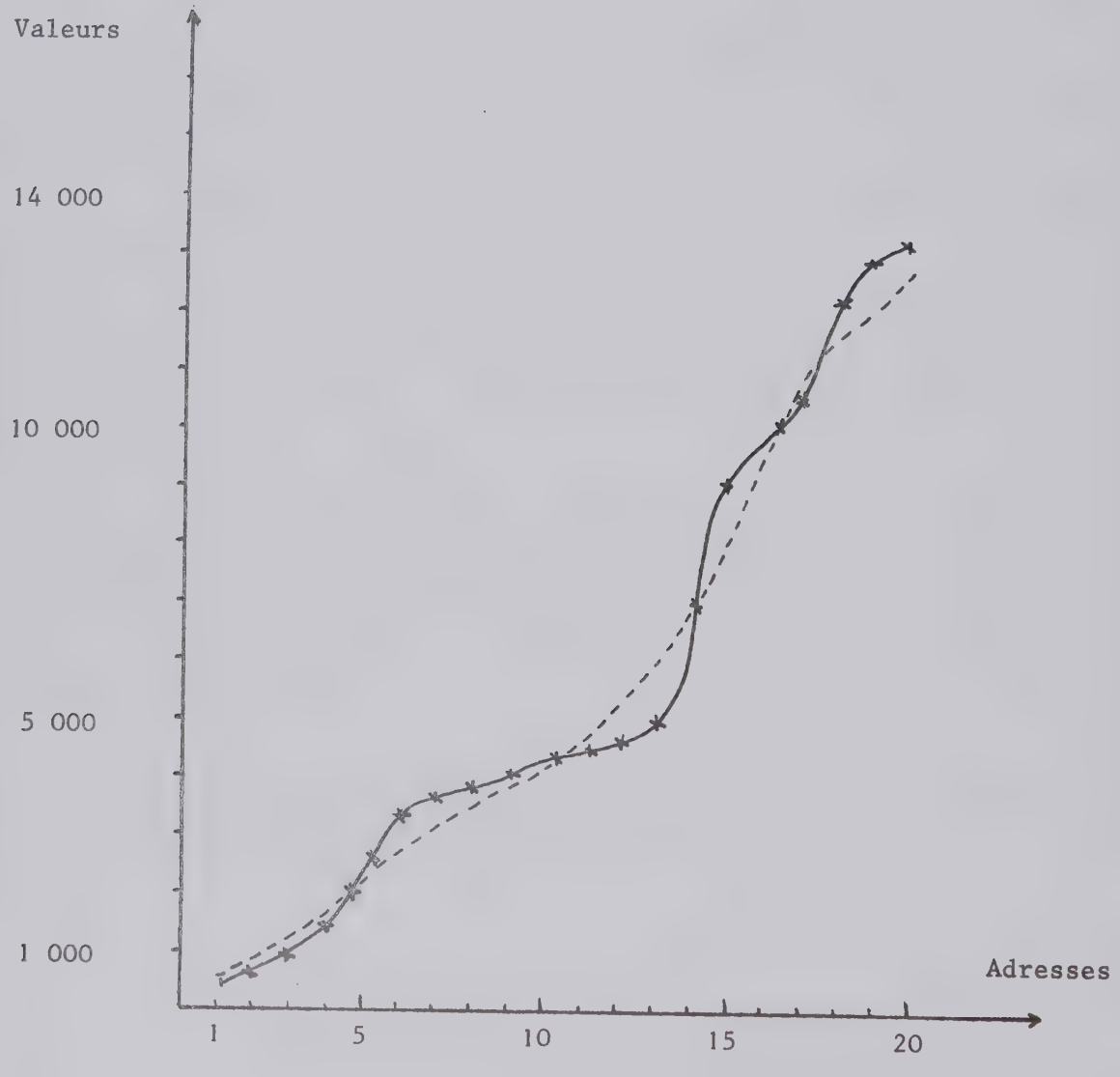
X = 2 000	Mémoire totale = 4 000
M = 10 %	16 150 entrées/sorties

Avec le tri fusion utilisé dans les mêmes conditions (S = 1) on aurait 2X(K + 1) entrées-sorties.

Soit 12 x 4 000 = 48 000 E/S

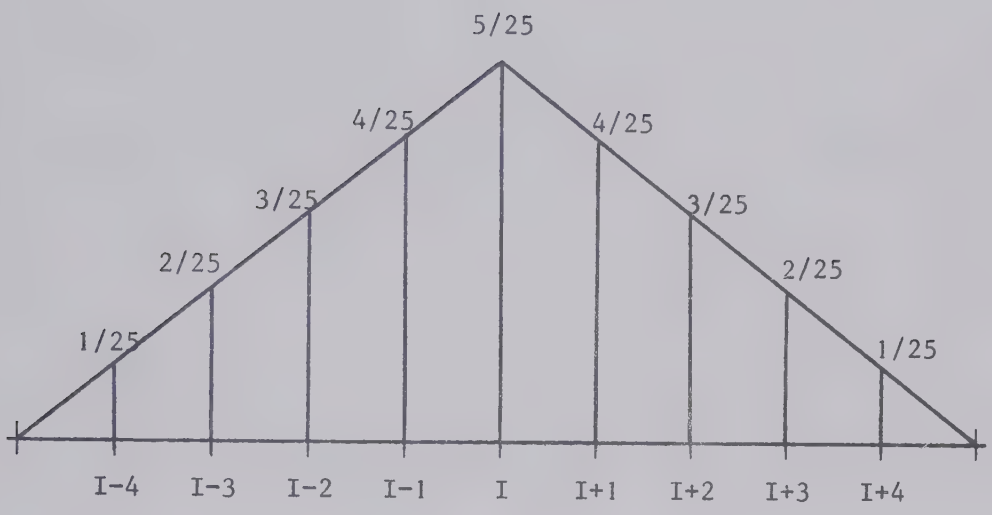
Soit 3 fois plus que pour la méthode approximative

Si on trace la courbe de rangement, on observe que cette courbe présente de fortes irrégularités (trait plein).



Ceci nous a donné l'idée de lisser la courbe (trait discontinu)

Le lissage est obtenu avec le filtre linéaire suivant :



Les résultats sont après lissage de 7 967 E/S (au lieu de 16 150).

Le minimum théorique s'il n'y avait pas de collisions serait

$$2\,000 + 2 \times 2\,000 = 6\,000$$

lecture des nombres	lecture de l'adresse et écriture à cette adresse

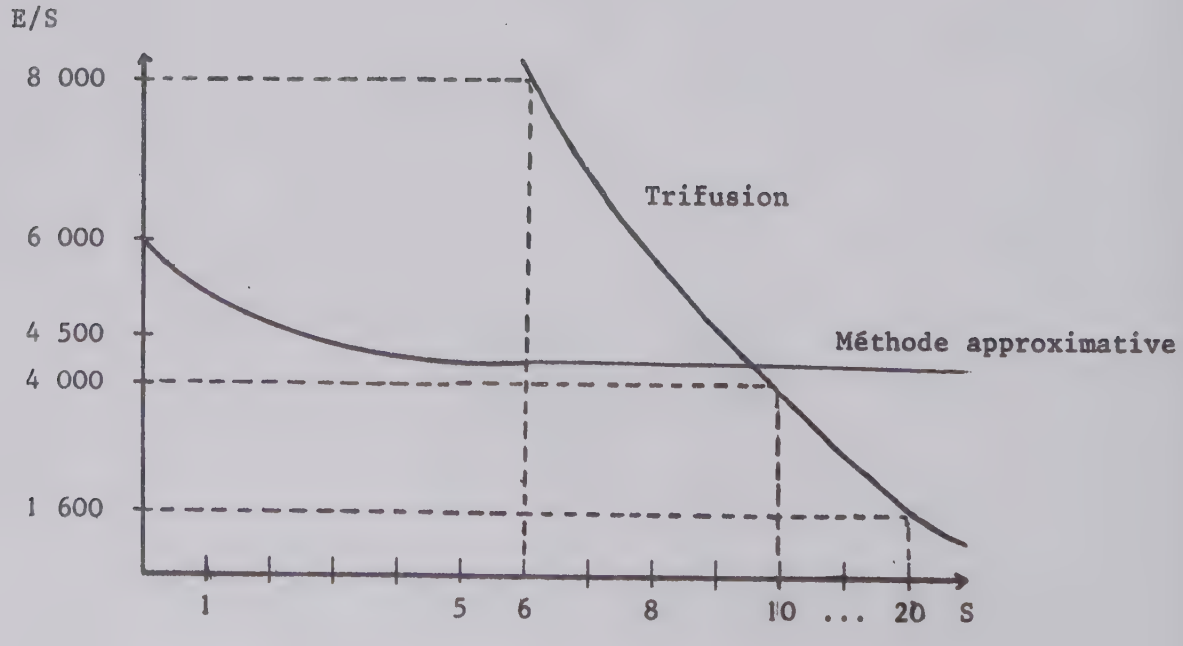
b. Utilisation de 2 zones tampons

On utilise une zone tampon pour lire les nombres à trier et une pour ranger les valeurs sur disque. L'algorithme de rangement utilisé consiste à essayer de rester dans la zone tampon lue. Ainsi, la recherche s'effectue dans un même bloc. Chaque nombre à ranger possède ainsi de fortes chances de ne nécessiter que 2 entrées-sorties disque. D'autre part, on n'écrit la zone tampon sur disque que si le nombre suivant doit être à une adresse non située sur la zone tampon encore en mémoire centrale. On a X/S blocs on a donc 1/(X/S) chances d'avoir 2 nombres consécutifs situés sur un même bloc.

Il faut donc cette fois

$$X/S + 2X(1 - S/X) \quad \text{entrées-sorties}$$

La comparaison avec le tri-fusion est donnée par les courbes suivantes :



La méthode approximative est donc meilleure que le trifusion pour de très petites valeurs de S. Ce sera le cas par exemple si les articles sont très longs.

Nous avons essayé d'améliorer la méthode approximative pour des zones tampons plus grandes. La méthode de trifusion utilisant trois zones tampons, nous en utiliserons trois dans les deux méthodes.

b. Utilisation de trois zones tampons

Au lieu de ranger chaque nombre, un à un sur le disque, nous allons les regrouper et nous les rangerons uniquement lorsque nous en aurons un certain nombre appartenant à un même bloc. Pour cela, nous allons réserver dans le troisième buffer un certain nombre de places pour chaque bloc. On a P/S blocs (P est la mémoire externe totale utilisée) donc chaque bloc est représenté par $S/(P/S) = S^2/P$ places mémoires. Il faut donc :

$$S \geqslant \sqrt{P}$$

On lit les valeurs, on calcule leur adresse et on les place dans le buffer dans la partie réservée au bloc contenant cette adresse. Lorsque cette partie est pleine et que l'article en cours appartient à ce bloc, on vide tous les articles de ce bloc soit $S^2/(P + 1)$ articles. On a divisé par conséquent les $2X$ entrées-sorties par $S^2/(P + 1)$ il faut donc au total

(1) $$X/S + 2X(S^2/(P + 1) \text{ entrées-sorties}$$

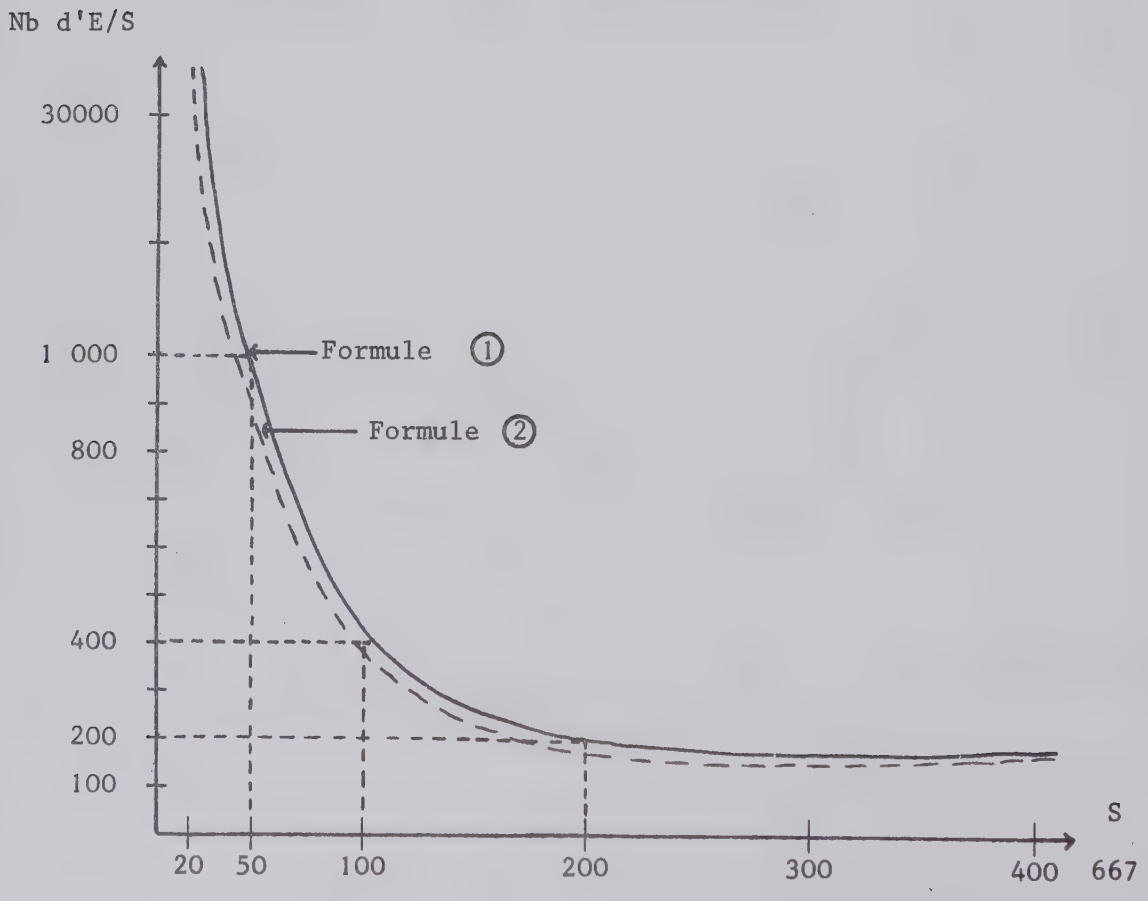
On peut encore améliorer cette méthode de la façon suivante : on lit k fois le fichier, à chaque passage on ne s'occupe que d'un certain nombre de blocs ($\frac{1}{k}$ du nombre total de bloc, soit $1/k \times P/S$ blocs).

Ainsi chaque bloc est représenté par kS^2/P places, la condition devient $k \geqslant P/S^2$ ou $S \geqslant \sqrt{\frac{P}{k}}$.

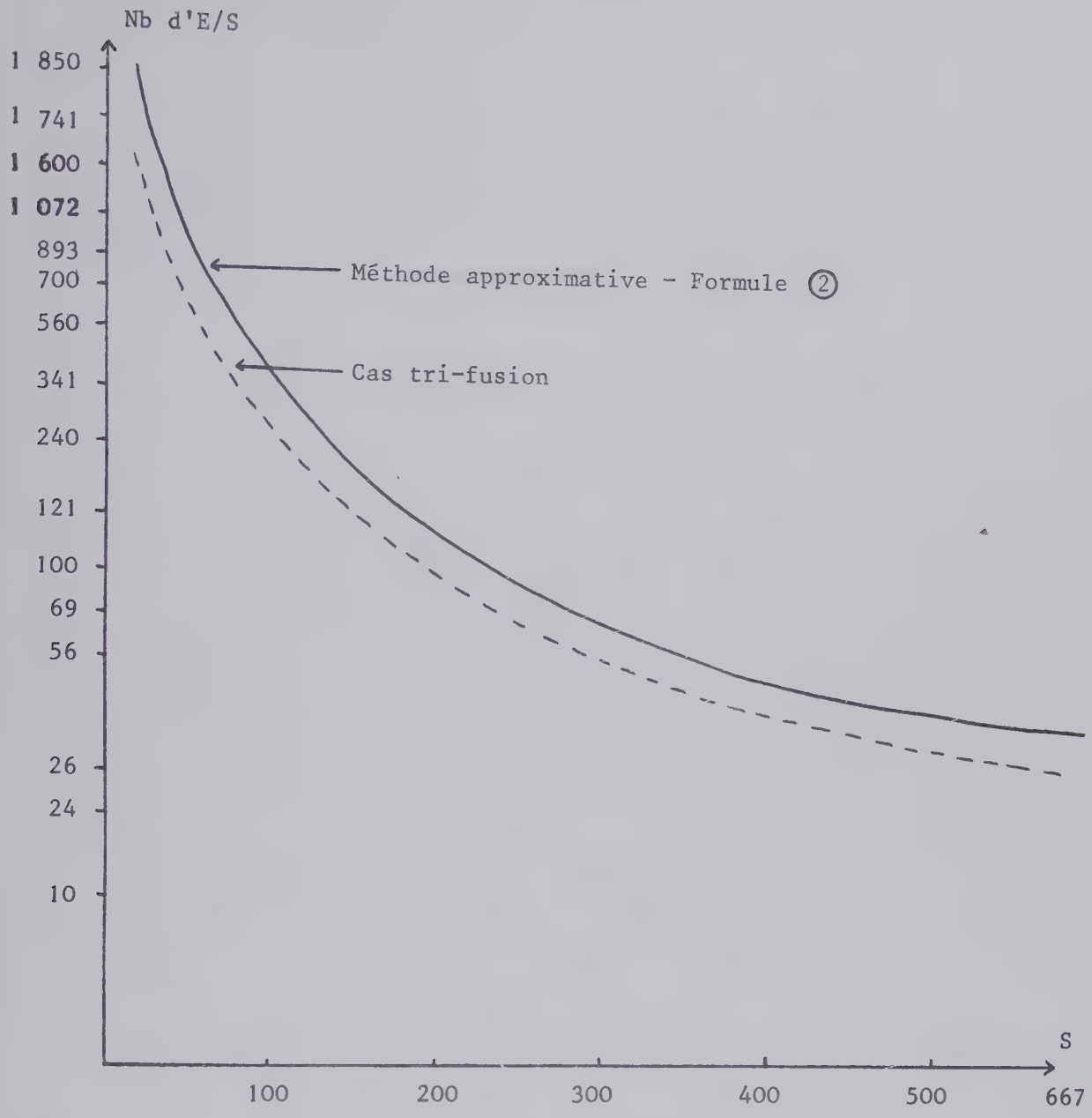
On a maintenant besoin de

(2) $\boxed{k \times X/S + 2X/((k\ S^2/P) + 1) \text{ entrées-sorties}}$

Il suffit de calculer k pour que cette expression soit **minimale** compte tenu de la condition $k \geqslant P/S^2$.



Courbes de comparaison formule ① → formule ②



Courbes de comparaison Trifusion ↔ formule (2)

En conclusion, la méthode approximative est donc meilleure que le trifusion pour un facteur de groupage faible ($S \leqslant 10$) et reste comparable ensuite. Il est surtout à noter que la méthode approximative permet un tri direct sans attendre l'ensemble du fichier pour commencer, une utilisation de certaines valeurs en cours de tri et une plus grande souplesse de modifications ensuite. Ce tri est à rapprocher de l'organisation séquentielle indexée utilisée couramment en gestion.

Les tableaux qui suivent donnent des résultats expérimentaux obtenus en faisant varier divers paramètres. Les nombres entre parenthèses représentent l'écart type.

X	M	P	écriture	lecture	remplis.	Nb d'E/S
200	20	400	18(5,9)	18(6,1)	0,5(0,18)	361
400	40	800	35,5(13,7)	35,5(13,8)	0,5(0,19)	710
600	60	1 200	54,9(12,4)	54,9(12,3)	0,5(0,12)	1 098
800	80	1 600	74(11,4)	74,1(11,3)	0,5(0,08)	1 481
1 000	100	2 000	89,9(16,9)	89,9(17,1)	0,5(0,10)	1 798
1 200	120	2 400	109(17,3)	109(17,2)	0,5(0,09)	2 184
1 400	140	2 800	126(13)	126(13)	0,5(0,05)	2 528
1 600	160	3 200	145(14)	145(14)	0,5(0,05)	2 892
1 800	180	3 600	162(25)	162(25)	0,5(0,08)	3 246
2 000	200	4 000	182(21)	182(21)	0,5(0,06)	3 646

Tableau 1

2 zones tampons. On fait varier X de 200 à 2 000, M et P conservant le même rapport. On a trois variables aléatoires :

- le nombre d'écriture disque
- le nombre de lecture disque
- le coefficient de remplissage d'un bloc.

On constate bien que le coefficient global de remplissage est de 50 % et qu'il varie faiblement suivant les blocs. L'écart type étant bien meilleur pour X proche de 2 000. Ceci provient du fait que 200 nombres sont plus représentatifs de 2 000 que 20 de 200.

X	M	P	écriture	lecture	remplis.	Nb d'E/S
2 000	200	4 000	182(21)	182(21)	0,5(0,06)	3 646
2 000	100	4 000	180(32)	180(32)	0,5(0,09)	3 607
2 000	50	4 000	182(40)	182(40)	0,5(0,11)	3 644
2 000	20	4 000	179(54)	179(54)	0,5(0,16)	3 580
2 000	10	4 000	196(118)	196(118)	0,5(0,25)	3 929
2 000	5	4 000	temps machine trop long			

Tableau 2

On conserve X fixe et on fait varier M (nombre d'échantillons représentatifs).

X	M	P	écriture	lecture	remplis.	Nb d'E/S
2 000	200	4 000	99(39)	99(39)	0,5(0,17)	3 967
2 000	100	4 000	101(46)	101(46)	0,5(0,18)	4 068
2 000	50	4 000	104(60)	104(60)	0,5(0,22)	4 194
2 000	30	4 000	107(68)	107(68)	0,5(0,22)	4 283
2 000	20	4 000	117(96)	117(96)	0,5(0,27)	4 699
2 000	10	4 000	131(119)	131(119)	0,5(0,30)	5 271
2 000	5	4 000	temps machine trop long			

Tableau 3

Ce tableau est obtenu de la même façon que le précédent mais la loi uniforme donnant les nombres de façon aléatoire est modifiée de façon à devoir deux points d'accumulation.

Tableau 4

S	écriture	lecture	remplis.	Nb d'E/S
20	5,2(1,5)	5,2(1,5)	0,5(0,15)	3 074
30	5,3(1,4)	5,3(1,4)	0,5(0,13)	2 020
50	4 (0,8)	4 (0,8)	0,5(0,10)	1 045
100	2,3(0,5)	2,3(0,5)	0,5(0,07)	387
200	2,4(0,5)	2,4(0,5)	0,5(0,04)	147
300	1,3(0,4)	1,2(0,4)	0,5(0,09)	84
400	1,2(0,4)	1,2(0,4)	0,5(0,01)	49
600	1,4(0,4)	1,3(0,4)	0,5(0,06)	35
667	1,3(0,4)	1,1(0,3)	0,5(0,16)	24

On utilise trois zones tampons avec stockage intermédiaire avant de ranger sur disque. X = 2 000 M = 2 P = 4 000.

S	écriture	lecture	remplis.	Nb d'E/S
20	6,8(1,8)	6,8(1,8)	0,66(0,17)	2 850
30	5,4(1,2)	5,3(1,2)	0,66(0,15)	1 741
50	3,4(0,6)	3,4(0,6)	0,66(0,11)	893
100	2,7(0,47)	2,3(0,46)	0,66(0,08)	341
200	2,4(0,47)	2,3(0,45)	0,66(0,05)	121
300	2,1(0,28)	2,0(0,42)	0,66(0,02)	69
400	2,0(0,66)	1,9(0,73)	0,66(0,11)	46
667	1,2(0,36)	1,0(0,30)	0,59(0,12)	26

Tableau 5

Même chose : X = 2 000 M = 2 P = 4 000

S	écriture	lecture	remplis.	Nb d'E/S
20	5,2(1,7)	5,2(1,7)	0,49(0,17)	3 078
34	6 (1,9)	6 (1,9)	0,50(0,16)	1 824
50	4,0(1,1)	4,0(1,1)	0,50(0,14)	1 031
100	2,3(0,56)	2,3(0,55)	0,50(0,12)	387
334	1,3(0,45)	1,2(0,41)	0,50(0,09)	67
400	1,4(0,63)	1,3(0,43)	0,50(0,06)	52
667	1,3(0,69)	1,2(0,34)	0,49(0,05)	24

Tableau 6

Même chose : X = 2 000 M = 2 P = 4 000

ANNEXE B

ANNEXE B

COURBES RESULTATS CONCERNANT LA METHODE APPROXIMATIVE

1. Seules, les fonctions d'approximation g_1 et g_3 décrites dans l'annexe C ont été utilisées dans nos essais. Les tableaux 7, 8, 9 et 10 suivants donnent l'évolution des coefficients β et γ suivant α et le nombre d'échantillons choisis (le nombre d'éléments du fichier est de 2 000). Les courbes 11 et 12 représentent ces tableaux avec g_1 et avec g_3.

	M = 200		M = 400		M = 800	
α	γ	β	γ	β	γ	β
0,1	1,02	1,02	-	-	-	-
0,2	1,10	1,08	1,05	1,05	-	-
0,3	1,23	1,14	1,13	1,10	-	-
0,4	1,45	1,24	1,30	1,17	1,09	1,09
0,5	1,84	1,39	1,51	1,26	1,20	1,14
0,6	2,60	1,67	1,87	1,40	1,38	1,28
0,7	3,76	2,08	2,42	1,59	1,66	1,31
0,8	7,72	3,18	3,26	1,88	2,19	1,46
0,9	19,86	6,73	6,09	2,49	3,49	1,78

Tableau 7 - Fonction "g_1"

Tableau 8 - Fonction "g_3"

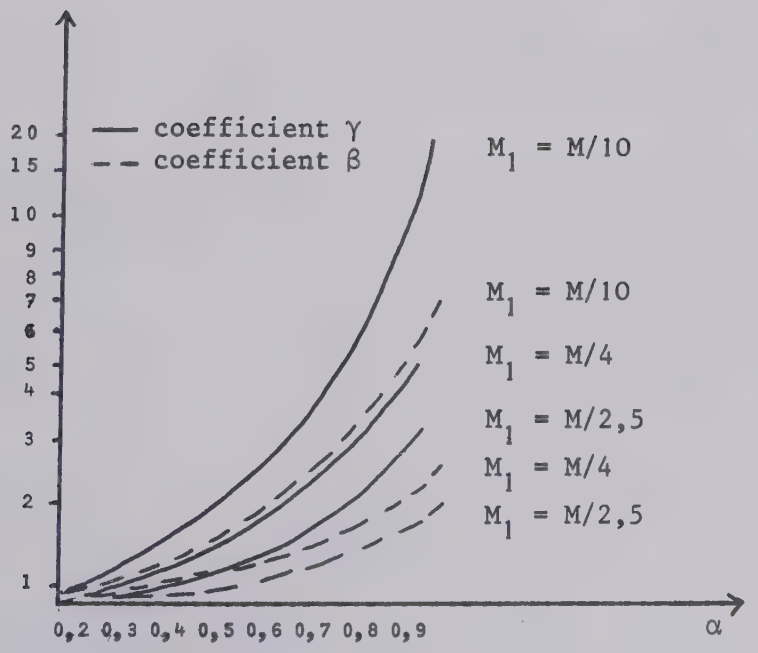
	M = 200		M = 400		M = 800	
α	γ	β	γ	β	γ	β
0,1	-	-	-	-	-	-
0,2	1,10	1,07	1,04	1,04	-	-
0,3	1,27	1,16	1,14	1,10	-	-
0,4	1,55	1,28	1,36	1,19	1,08	1,08
0,5	2,06	1,48	1,62	1,31	1,21	1,14
0,6	2,90	1,83	2,00	1,47	1,42	1,23
0,7	4,05	2,29	2,60	1,69	1,71	1,34
0,8	7,74	3,27	3,47	2,00	2,23	1,50
0,9	15,96	5,29	6,33	2,63	3,54	1,83

Tableau 9 - Fonction "g_1" (1 à 10 ù d'échantillons)

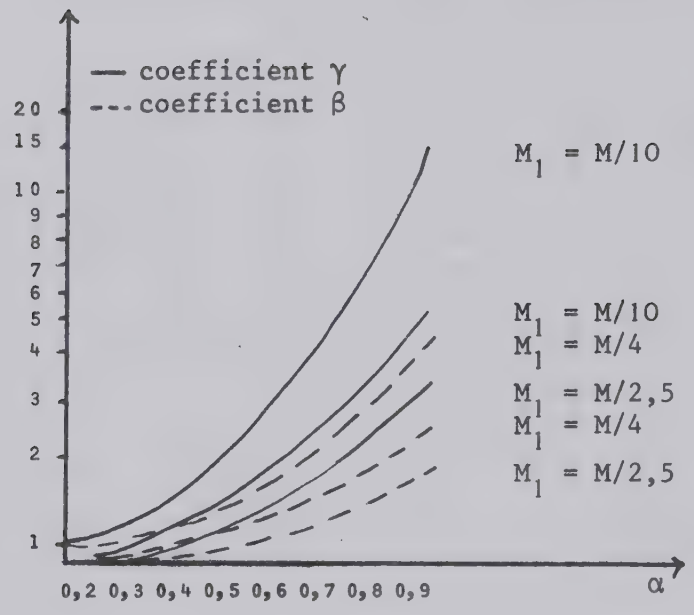
	M = 40		M = 80		M = 120		M = 160	
α	γ	β	γ	β	γ	β	γ	β
0,1	1,45	1,48	1,35	1,35	1,31	1,32	1,25	1,27
0,2	2,71	2,34	1,80	1,62	1,67	1,52	1,57	1,43
0,3	4,17	3,26	2,54	2,07	2,05	1,78	1,91	1,67
0,4	5,34	4,03	3,16	2,47	2,58	2,06	2,37	1,92
0,5	10,34	5,89	5,54	3,38	4,20	2,69	4,25	2,65
0,6	18,52	9,19	9,72	5,05	7,17	3,77	6,87	3,77
0,7	25,9	13,09	15,07	7,36	10,26	5,18	9,31	5,04
0,8	42,87	17,93	36,31	11,74	16,18	7,28	15,00	6,89
0,9	66,01	24,78	85,13	13,28	28,97	10,63	26,89	9,90

Tableau 10 - Fonction "g_3" (1 à 10 échantillons)

	M = 40		M = 80		M = 120		M = 160	
α	γ	β	γ	β	γ	β	γ	β
0,1	1,50	1,54	1,38	1,37	1,34	1,32	1,27	1,28
0,2	2,76	2,29	1,96	1,72	1,81	1,54	1,69	1,50
0,3	4,36	3,25	2,90	2,26	2,36	1,92	2,17	1,79
0,4	5,73	4,10	3,67	2,74	3,16	2,32	2,86	2,15
0,5	10,60	6,04	6,17	3,73	4,98	3,09	4,44	2,84
0,6	18,08	9,18	9,61	5,23	7,90	4,21	6,85	3,88
0,7	25,35	12,88	13,98	7,07	11,48	5,66	9,72	5,19
0,8	43,27	17,46	28,12	10,00	18,54	7,92	16,28	7,15
0,9	65,99	23,81	49,83	15,55	31,99	11,45	28,74	10,41



Courbe 11 - Méthode approximative fonction g_1



Courbe 12 - Méthode approximative fonction g_3

2. Le tableau 14 donne l'influence d'un filtre du type de celui décrit dans le paragraphe 6-e. Ce tableau est à comparer avec le tableau 9.

Tableau 14 - Fonction "g_1" avec filtre

	M = 40		M = 80		M = 120		M = 160		M = 200	
α	γ	β	γ	β	γ	β	γ	β	γ	β
0,1	1,47	1,56	1,24	1,37	1,61	1,39	1,06	1,35	1,00	-
0,2	2,75	2,58	1,80	2,02	1,60	1,99	1,44	1,99	1,38	2,00
0,3	4,43	3,61	2,96	2,66	2,41	2,43	2,03	2,27	1,51	2,24
0,4	5,61	4,45	3,58	3,11	2,98	2,78	2,56	2,52	1,99	2,41
0,5	9,28	5,98	5,72	3,96	4,64	3,46	4,14	3,12	3,37	2,95
0,6	15,34	8,60	8,57	5,23	6,59	4,36	5,78	3,90	5,18	3,72
0,7	25,84	12,94	10,55	6,53	7,91	5,23	7,08	4,66	7,03	4,60
0,8	83,62	-	16,44	8,24	10,82	6,37	15,97	6,21	12,72	6,21
0,9	-	-	38,89	10,02	22,14	8,99	37,37	11,12	41,96	7,63

3. Les courbes 15 et 16 représentent des comparaisons entre la méthode approximative obtenue avec g_1 et g_3 et deux méthodes de "Hash Coding" HC1 et HC3

HC1 $h_i(x)$ = reste $(x/MM) + 7 * i$ Modulo MM

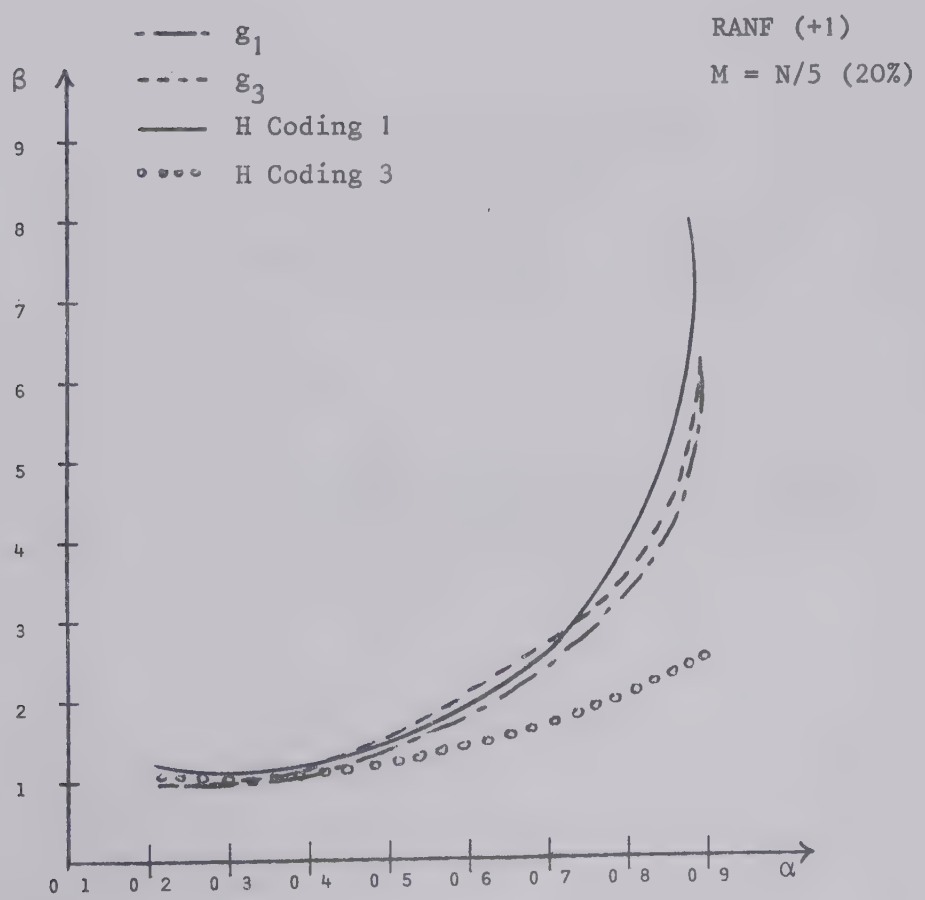
HC3 $h_i(x)$ = reste $(x/MM) + 7 * i + (x/MM) * i^2$ Modulo MM

MM est la taille de la mémoire.

On constate que la méthode approximative est comparable à la moins bonne des méthodes : HC1.

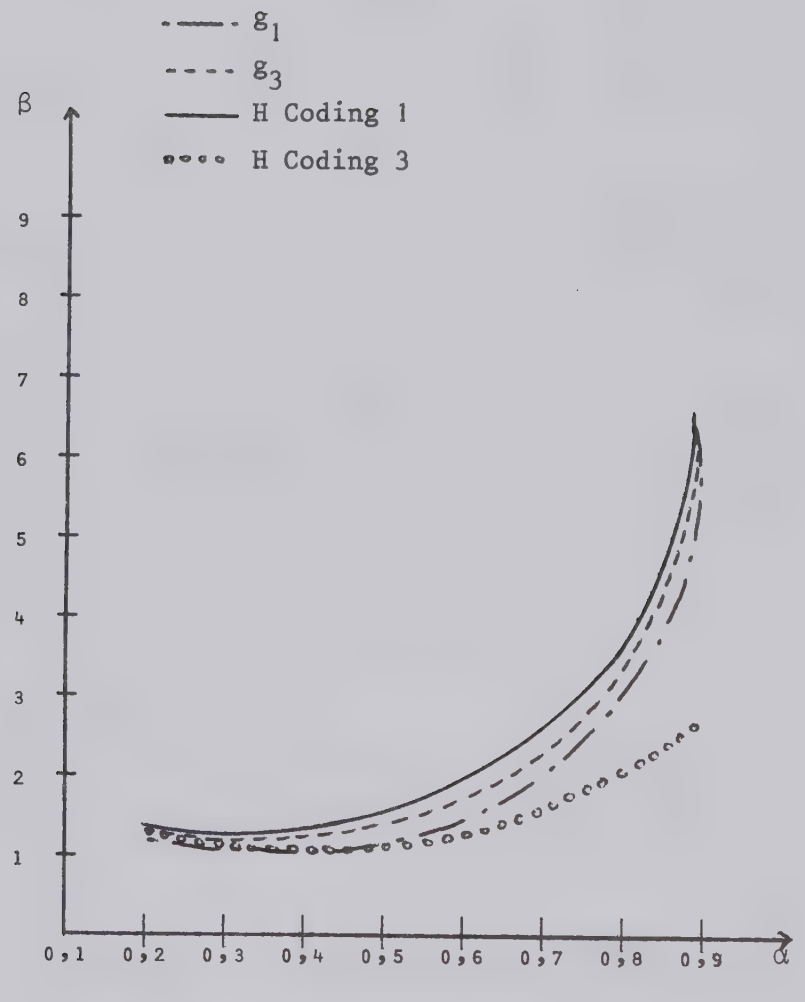
La courbe 16 se différencie de la courbe 15 par le fait que le fichier de nombres aléatoires est modifié de façon à ce que la loi correspondante soit plus uniforme mais présente un palier.

La transformation est donnée par $y = a(1 + (\frac{3x}{a-1})^3)$.



Courbe 15 - Comparaison FG1 - FG3 - H Coding (2 méthodes)

Fonction RANF modifiée



Courbe 16 - Comparaison des méthodes ($M = \frac{N}{S}$ (20%))

ANNEXE C

ANNEXE C

FONCTIONS D'APPROXIMATION

La recherche de la méthode d'approximation est soumise à plusieurs contraintes :

- la méthode doit être locale. Il n'est pas concevable dans les applications que nous avons choisies de parcourir à chaque étape l'ensemble des échantillons.

- les fonctions utilisées doivent être aisément calculables par ordinateur.

Tenant compte de ces éléments nous avons mis en évidence un ensemble de fonctions d'interpolation qui ne sont pas toutes continues. Cet ensemble comprend des fonctions splines périodiques et certaines fonctions découvertes par SCHOENBERG (1946).

Les problèmes est de trouver une fonction f_n telle que f_n représente la fonction f avec une "borne" précision.

$$f_n(x) = \sum_{p \in Z} f(p\,u)\; g_n\left(\frac{x}{u} - p\right)$$

u est le pas d'échantillonnage.

Les conditions sur g_n qui proviennent de nos contraintes sont :

1. Le support de g_n est limité et plus grand que $n + 1$.
2. g_n est constitué d'arcs de polynômes de degré au plus égaux à n.
3. g_n est une fonction symétrique.

L'idée est d'utiliser des échantillons proches de la valeur x et d'en utiliser un nombre réduit fonction de n. Si $n = 0$ on n'utilise que l'échantillon le plus proche ; si $n = 1$ les deux échantillons encadrant x sont utilisés. Si $n = 2$ on utilise les deux précédents et le troisième plus proche, etc...

En chaque point pu on peut effectuer un développement en série de Taylor

$$f(pu) = \sum_{k=0}^{n-1} \frac{(pu) - x)^k}{k\ !}\ f^{(k)}(x) + \frac{(pu - x)^n}{n\ !}\ f^{(n)}(C_{n,p})$$

Si on multiplie par $g_n(\frac{x}{u} - p)$ et que l'on somme sur p on obtient :

$$\sum_{p \in Z} f(pu)\ g_n(\frac{x}{u} - p) = \sum_{k=0}^{n-1} (-1)^k \frac{u^k}{k\ !} \sum_{p \in Z} (\frac{x}{u} - p)^k\ g_n(\frac{x}{u} - p)\ f^{(k)}(x) + R_n^{(x)}$$

On impose alors une condition supplémentaire sur les fonctions g_n :

$$(1) \qquad \sum_{p \in Z} (\frac{x}{u} - p)^k\ g_n(\frac{x}{u} - p) = \delta_0^k \qquad k \in [0\ \ n-1]$$

et on a alors $\qquad f_n(x) = f(x) + R_n(x)$

La formule donnant $R_n(x)$ permettant une mesure de l'erreur commise par interpolation.

Le système d'équations (1) est un système de Cauchy d'ordre n. Il se résoud aisément et donne pour solution si on suppose le support de g_n symétrique et de longueur n :

$$g_n(x) = \frac{(-1)^n}{n\,!}\,(-1)^i\,\binom{n}{i}\prod_{\substack{j=1\\ j\neq 0}}^{n-i}(x-j)$$

Pour $n/2^{-i} < x < n+1/2^{-i}$

Les figures 17, 18, 19 et 20 donnent l'allure des fonctions g_0, g_1, g_2, g_3. On remarquera que g_0 donne l'interpolation en marche d'escalier et g_1 l'interpolation linéaire. On remarquera aussi que g_0 et g_2 ne sont pas continus et que g_3 quoique peu différent de g_1 donne des résultats 1 000 fois meilleurs dans l'approximation de fonctions telles un sinus.

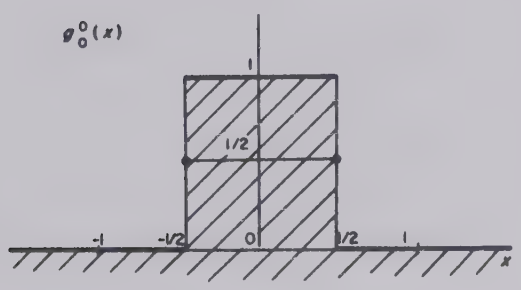


Fig. 17

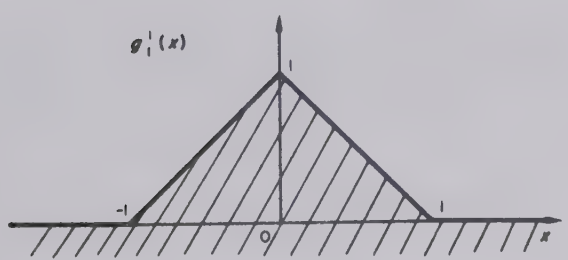


Fig. 18

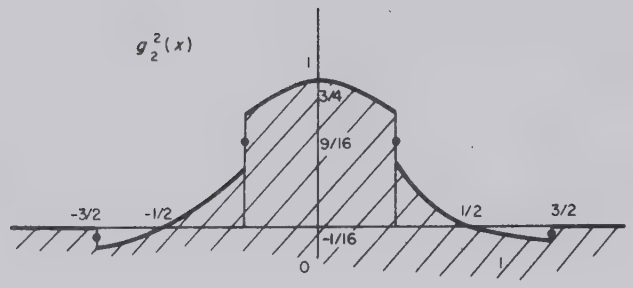


Fig. 19

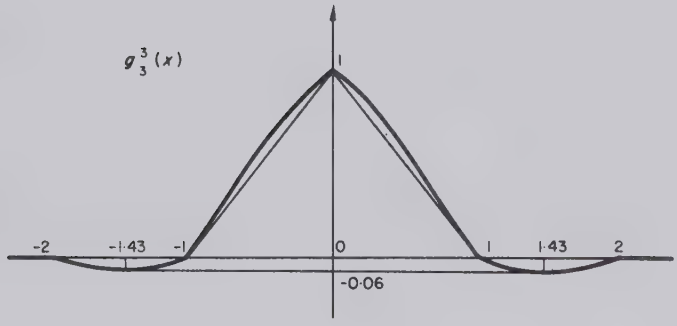


Fig. 20

BIBLIOGRAPHIE

BIBLIOGRAPHIE

[1] *AFIRO*

Section 2. Colloque sur les langages de listes
organisé à Grenoble, 21-25 Mars 1967.

[2] *BACHMAN and WILLIAMS*

A general purpose programming system for random access memory
AFIPS Conf. proceedings, vol. 26, part. 1, 1964.

[3] *BAEKER H. D.*

Mapped list structures
CACM, vol. 6, p. 435 (Août 1963).

[4] *BAZERQUE G.*

Implantation des arborescences : applications aux tableaux
R. I. R. O., 4ème année, B-3, 1970, 27-107.

[5] *BELL J. R., KAMAN C. H.*

The linear quotient Hash Code
Com. ACM 13, 11 (Nov. 1970) pp. 675-677.

[6] *BELL J. R.*

The quadratic quotient method
Com. ACM - Fev. 1970.

[7] *BERGE C.*

Graphes et Hypergraphes
Ch. 3 - Dunod.

[8] *BERKELEY E., BOBROW D.*

The programming language LISP : its operation and applications
The MIT Press, 1966.

[9] *BARNARD J.*

Les codifications et les machines à mémoires adressables
Chiffres, 1963, n° 3.

[10] *BOBROW D. G. and RAPHAEL B.*

A comparison of list processing languages
CACM, vol. 2, p. 4 (Février 1959).

[11] *BRASSEUR M. et COHEN J.*

Algorithmes d'analyse syntaxique pour languages "context-free"
RFTI Chiffres, vol. 8, n° 2, 1965, pp. 95-119, vol. 8, n° 3, 1965, pp. 195-213.

[12] *BRENT R. P.*

Reducing the retrieval time of scatter storage techniques
Com. ACM (Februar 1973), pp. 105-109.

[13] *BROISE P.*

Organisation en machine des données structurées
R. I. R. O., 4ème année, B-3, 1970, 3-26.

[14] *BROUDIN Y.*

Le traitement des listes en programmation, SLIP : langage de listes symétriques, Applications
Rapport CEA R 3040, Paris 1966.

[15] *CAGNAT J. M.*

Structures de représentation des données en ordinateur, application aux traitements graphiques
Thèse 3ème cycle, Grenoble 1971.

[16] *CARR J. W.*

Recursive subscripting compilers and list-type memories
CACM, vol. 2, p. 4 (Février 1959).

[17] *CASSO J. M.*

Méthode de rangement en mémoire à l'aide des fonctions g_1 et g_3
1971 - Rapport L de P.

[18] *CHATY G.*

Sur les traitements de listes
Rapport I. de P.

[19] *CHENEY C. J.*

A non recursive list compacting algorithm
CACM, vol. 13, n° 11, nov. 1970, pp. 677-678.

[20] *CLAVIER J., NIQUIL M., COFFINET G. et BEHR F.*

Théorie et technique de la transmission de données
Masson - Paris 1972.

[21] *CLEVERDON C. W.*

Identification of criteria for evaluation of operational information retrieval system
Cranfield College of Aeraunotics, England - Nov. 1964.

[22] *COHEN J.*

A use of faxt and slow memories in list processing languages
IMAG, Grenoble, Aôut 1965.

[23] *COHEN J. et NGUYEN HUU DUNG*

Définition de procédures LISP en ALGOL et exemples d'utilisation
RFTI Chiffres, vol. 8, n° 4, 1965, pp. 271-293.

[24] *COMFORT W. T.*

Multi-word list items
CACM, vol. 7, p. 357 (Juin 1964).

[25] *CONWAY R. W., DELFAUSSE J. J., MAXWELL W. L. and WALKER W. E.*

CLP-the Cornell list processor
CACM, vol. 8, p. 215 (avril 1965).

[26] *CRESPI-REGHIZZI S. and MORPURGO R.*

A language for treating graphs
CACM, vol. 13, n° 5, May 1970.

[27] *CRESSON R., DETHOOR J. M. and GROBOILLOT J. L.*

Les principaux types d'organisation de fichiers
Revue Informatique et Festion, n° 23, Dec. 1970.

[28] *DEBRAINE P.*

Machines de traitement de l'Information, circuits et programmes
Masson - Paris.

[29] *DUMEY A. Y.*

Indexing for rapid random-acces memory
Computers and Automation 5, n° 12, pp. 6-8 (Dec. 1956).

[30] *EVANS A., PERLIS A. J. and VAN VOERHAN H.*

The use of threated lists in constructing a combined ALGOL and machine like assembly processor
CACM, vol. 4, p. 36 (Janvier 1961).

[31] *EVANS T., COHEN J., TRILLING M. SIRET Y. et LECARME O.*

Publications à l'occasion du symposium de traitement de listes à l'IMAG
Grenoble, Mars 1966.

[32] *FENICHEL R. R. and YOCHELSON J. C.*

A LISP garbage-collector for memory computer systems
CACM, vol. 12, n° 11, Nov. 1969, pp. 611-612.

[33] *FLORES I.*

Computer Sorting
1969

[34] *FOSTER J. M.*

List processing
Macdonald and Co., LTD, Londres, 1967 (Traduction française: Dunod, Paris 1970).

[35] *GELERTNER H.*

A FORTRAN compiled list processing language
JACM, vol. 7, p. 87 (avril 1960).

[36] *GUIHO G.*

Sur l'étude des collisions dans les méthodes de Hash Coding
C. R. A. S. Série A - pp. 580-583 - 14 Fév. 1972.

[37] *GUIHO G. et ROTH S.*

Une méthode de résolution des collisions dans le Hash Coding
Congrès AFCET - 1972 (Grenoble).

[38] *GUIHO G. D.*

Gain théorique de mémoire obtenu à l'aide d'un algorithme de prétraitement
CR. Acad Sciences (Janvier 1973) pp. 65-68.

[39] *GUIHO G. D.*

Méthodes de recherche en mémoire
Automatisme (Mai 1973) pp. 194-199.

[40] *GUIHO G. D.*

Méthodes de recherche en mémoire et étude d'optimisation
Colloque International sur les mémoires - Paris (Octobre 1973).

[41] *HANAN M. et PALERMO F. P.*

An application of coding theory to a file addressing problem
IBM J. R. and D.7, 2 Apr. 1963, pp. 127-129.

[42] *HANSEN W. J.*

Compact list representation : definition, Garbage collection and system implementation
CACM, vol. 12, n° 9, 1969.

[43] *HARRAND Y.*

Traitement des files et des listes
Dunod, Paris, 1967.

[44] *HOUDAS et HORVAT*

Thèse de 3ème cycle - Juin 1970 - Institut de Programmation - Paris.

[45] *HSIAO D. and HARARY F.*

A formal system for information retrieval of files
CACM, vol. 13, n° 2, February 1970.

[46] *IVERSON K. E., BROOKS F. P.*

Automatic data processing
Wiley 1969.

[47] *JENKINS D. P.*

List programming, in Introduction to system programming
Academic Press, 1964.

[48] *JOHNSON L. R.*

Indirect chaining for addressing on secondary keys
Com. ACM - May 1961.

[49] *JONES B.*

A variation on Sorting by address calculation
Com. ACM 13 (Feb. 1970) pp. 105-107.

[50] *KNOWLTON K. C.*

A programmer's description of L6
CACM, vol. 9 (august 1966).

[51] *KNUTH D. E.*

The art of computer programming, tome I, Fundamental algorithms
Addison Wesley Reading Mass, 1968.

[52] *KNUTH D. E.*

Fundamental algorithms
p. 305 - Addison - Wesley.

[53] *LUM, YUEN and DODD*

KEY TO ADDRESS transform techniques
Com. of th ACM - April 1971.

[54] *LOUIT G.*

Algorithmes de tri
Dunod 1971.

[55] *McCARTHY J.*

Recursive functions of symbolics expressions and their computation by machine
Part. 1, CACM, Vol. 3, p. 184 (april 1960).

[56] *McCARTHY J. et al.*

LISP 1.5
Programmers manual - MIT Press, 1962.

[57] *McILROY M. D.*

A variant method of file searding
Com. ACM 6 (Janvier 1963) p. 101.

[58] *MACCHI C.*

Thèse complémentaire
Institut de Programmation (Juin 1972).

[59] *MANASFI M. S.*

Correction par ordinateur des altérations phonétiques des noms français. Calcul d'adresses et recherche en table
Thèse de 3ème cycle - Paris 1970.

[60] *MANNA Z.*

Formalization of properties of Programs
Stanford artificial intelligence report
Projet Memo AI.64, Juillet 68.

[61] *MAURER W. D.*

An improved Hash code for scatter storage
Com. of the ACM - Janvier 1968.

[62] *MORRIS R.*

Scatter storage techniques
Com. of the ACM - Janvier 1968.

[63] *MULLERY A. P. and SCHAUER R. F. ADAM*

A problem oriented symbol processor
Proc. AFIPS 1963, Spring Joint Computer Conference

[64] *NEWELL A. et al.*

Information Processing Language V, Manual Sections I and II
Rand. Corp. p. 1918 (Mars 1960).

[65] *NEWELL A. and TONGE F.*

An introduction to Information Processing Language V
CACM, vol. 3, p. 205 (april 1960).

[66] *NEWELL A.*

Documentation of IPL V
CACM, vol. 6, p. 86 (Mars 1963).

[67] *PERLIS A. J. and THORNTON C.*

Symbol manipulation by threated lists
CACM, vol. 3, p. 195 (avril 1960).

[68] *PETERSON W. W.*

Addressing for random-access storage
IBM Journal Res. and Dev. - April 1957.

[69] *PETERSON W. W.*

Error correcting codes
M. I. T. Press. - Cambridge, Mass 1961.

[70] *PICARD C. F.*

Panorama des liaisons entre graphes et ordinateurs
Théorie des graphes ICC (P. ROSENTHIEL ed.) Dunod, Gordon and Breach 1967, pp. 301-314.

[71] *PICARD C. F.*

Graphe et questionnaire
Gauthier-Villars, Paris 1972.

[72] *Proceedings of the CERN*

Computing and Data processing school
Genève, 1971.

[73] *RADKE C. E.*

The use of quadratic residue research
Com. ACM, 13 Février 1970, pp. 103-105.

[74] *Research Lab. of Electronics and MIT Computation Center*

Introduction to COMIT programming
MIT Press, 1961.

[75] *RIBBENS D.*

Programmation non numérique, LISP 1.5
Dunod - Paris - 1969.

[76] *ROCHE C.*

Thèse d'Etat (1973).

[77] *ROUBINE E.*

Introduction à la théorie de la communication
Tome III, Masson - Paris - 1970.

[78] *SALTON G.*

The SMART retrieval system
Prentice Hall - 1971.

[79] *SALTON G.*

Automatic Information organisation and retrieval
McGRAW HILL - 1968.

[80] *SCHAY G. and RAVER N.*

A method for key to address transformation
IBM J. R. and D. 7., 2 Apr. 1963 - pp. 121-127.

[81] *SCHAY G. and SPRUTH W. G.*

Analysis of a file addressing method
Com. ACM - Aug. 1962.

[82] *SENKO M. E.*

Formatted File Organisation Techniques
Final Report IBM - May 16, 1967.

[83] *SCHOENBERG*

Contribution to the problem of approximation
Q, J Appl. Math. 4 (Part A) 49, 99.

[84] *SHANNON C. E. and WEAVER W. W.*

Mathematical Theory of Communication
University of Illinois Press, 1949.

[85] *SHAPIRO S. C.*

The list set generator : a construct for evaluating set expressions
CACM, vol. 13, n° 12, Dec. 1970.

[86] *SIMON J. C.*

Compilation
1972 - Institut de Programmation.

[87] *SIMON J. C. et GUIHO G.*

Une nouvelle méthode d'établissement et d'utilisation de mémoires ordonnées
C. R. Acad. Sc. Paris, 271, 1970, pp. 1022-1025.

[88] *SIMON J. C. et GUIHO G.*

Picture representation and transformations by computer
Pattern Recognition Pergamon Press 1971, vol. 3, pp. 169-178.

[89] *SIMON J. C. et GUIHO G.*

On algorithms preserving neighbourhood to file and retrieve informations in a memory
Inter. J. of Comp. and Inform. Sciences. vol. 1, n° 1, pp. 3-15.

[90] *STARYNKEVITCH D.*

Une structure de machine à liste
RIRO, 1ère année, n° 3, 1967, pp. 13-31.

[91] *TABORY R.*

Premiers éléments d'un langage de programmation pour le traitement en ordinateur des graphes
IBM-France, Paris.

[92] *WEIZENBAUM J.*

Knotted list structures
CACM, vol. 5, p. 161 (Mars 1962).

[93] *WEIZENBAUM J.*

Symmetric list processor
CACM, vol. 6, p. 524 (Sept. 1963).

[94] *WIGINGTON R. L.*

A machine organization for a general purpose list processor
IEEE Trans. on Elect. - Comp. Dec. 1963.

[95] *WOODWARD P. M.*

List processing in Advances in programming and non numerical computation
Pergamon, 1966.

[96] *YNGLE V. H. COMIT*

CACM, vol. 6, p. 83 (Mars 1963).